NPS-SW-00-002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Engineering Automation for Reliable Software

Interim Progress Report (10/01/1999 – 09/30/2000)

By

Luqi

September 2000

20001204 063

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RADM David R. Ellison
Superintendent

Richard S. Elster
Provost

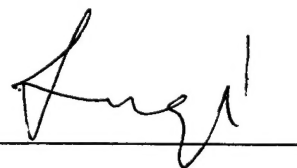This report was prepared for U.S. Army Research Office
and funded in part by the U.S. Army Research Office.

Prepared by:

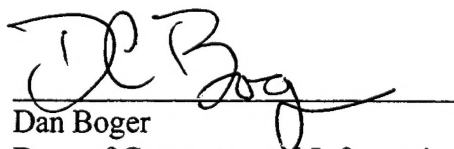Reviewed by:

Luqi
Professor, Computer Science

Luqi
Director, Software Engineering
Automation Center

Reviewed by:

Released by:

Dan Boger
Dean of Computer and Information Sciences
and Operations

D. W. Netzer
Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 9/30/2000	3. REPORT TYPE AND DATES COVERED Interim Progress Report 10/01/1999 – 09/30/2000

4. TITLE AND SUBTITLE
Engineering Automation for Reliable Software –
Interim Progress Report (10/01/1999 – 09/30/2000)

5. FUNDING NUMBERS
40473-MA-SP

6. AUTHOR(S)
Professor Luqi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Automation Center,
Naval Postgraduate School, Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-SW-00-002

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objective of our effort is to develop a scientific basis for producing reliable software that is also flexible and cost effective for the DoD distributed software domain. This objective addresses the long term goals of increasing the quality of service provided by complex systems while reducing development risks, costs, and time. Our work focuses on "wrap and glue" technology based on a domain specific distributed prototype model. The key to making the proposed approach reliable, flexible, and cost-effective is the automatic generation of glue and wrappers based on a designer's specification. The proposed "wrap and glue" approach allows system designers to concentrate on the difficult interoperability problems and defines solutions in terms of deeper and more difficult interoperability issues, while freeing designers from implementation details. Specific research areas for the proposed effort include technology enabling rapid prototyping, inference for design checking, automatic program generation, distributed real-time scheduling, wrapper and glue technology, and reliability assessment and improvement. The proposed technology will be integrated with past research results to enable a quantum leap forward in the state of the art for rapid prototyping.

14. SUBJECT TERMS
Rapid Prototyping, Lightweight Inference, Automatic Program Generation, Distributed Real-time Scheduling, Wrapper and Glue, Reliability Assessment, Interoperability, Heterogeneous System Integration

15. NUMBER OF PAGES
303

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION ON THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table of Contents

Interim Progress Report

Engineering Automation for Reliable Software

10/1/1999 – 9/30/2000

Luqi

List of Manuscripts:

M. Auguston, "Assertion Checker for the C Programming Language based on Computations over event traces", in *Proceedings of the Fourth International Workshop on Algorithmic and Automatic Debugging*, AADEBUG'2000, Munich, Germany, August 28-30, 2000. Also available on-line at http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html

M. Auguston, "Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars", in *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering*, Chicago, USA, July 6-8, 2000, pp.159-166.

M. Auguston, "Lightweight semantics models for program testing and debugging automation", in *Proceedings of 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario"*, Santa Margherita Ligure, Italy, June 13-16, 2000. Also available on-line at http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/

M. Auguston, "Building Program Behavior Models", submitted to the journal Applied Intelligence for the Special Issue on Spatial & Temporal Reasoning, 2000.

V. Berzins, "Static Analysis for Program Generation Templates", in *Proceedings of 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario"*, Santa Margherita Ligure, Italy, June 13-16, 2000. Also available on-line at http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/

N. Chaki, and S. Bhattacharya, "High Level Net Models : A Tool for Fault-Detection in Multistage Interconnection Network", in *Proceedings of the IEEE Region 10 Conference (TENCON) 2000*, Kuala Lumpur, Malaysia, September 24-27, 2000.

N. Cheng, V. Berzins, Luqi, and S. Bhattacharya, "Automated Generation of Wrappers for Interoperability", in *Proceedings of the 2000 Command and Control Research and Technology Symposium*, Monterey, CA, June 26-28, 2000.

J. Ge, V. Berzins, and Luqi, "Computer Aided Prototyping in a Distributed Environment", in *Proceedings of the International Congress on Intelligent Systems and Applications(ISA'2000)*, to appear in *Proceedings of the Symposium on Interactive and Collaborative Computing*, Australia, December 12-15, 2000.

J. Gregory, D. Zhang and M. Lu, "A Framework for Natural Language Agents", in *Proceedings of Ninth International Conference on Intelligent Systems*, Louisville, Kentucky, June 2000, pp. 57-60.

O. Kiselyou, "Implementing Metcast in Scheme", in Proceedings of the Workshop on Scheme and Functional Programming, Montréal, September, 2000, pp. 23-25.

B. Kitchenham and N. Schneidewind, "Towards an Ontology of Software Maintenance", in *Journal of Software Maintenance: Research and Practice*, Vol. 11, 1999, pp. 365-389.

Luqi, V. Berzins, M. Shing, N. Nada and C. Eagle, "Computer Aided Prototyping System (CAPS) for Heterogeneous Systems Development and Integration", in *Proceedings of the 2000 Command and Control Research and Technology Symposium*, Monterey, CA, June 26-28, 2000.

Luqi and J.C. Nogueira, "A Risk Assessment Model for Evolutionary Software Projects", in Proceedings of 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000. Also available on-line at http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/

N. Nada, V. Berzins, and Luqi, "Prototyping Tool-Kit (APT)", in Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000), ICSE 2000, Limerick, Ireland, June 4-11, 2000.

S. Pramanik, S. Choudhury, N. Chaki, and S. Bhattacharya, "Conceptual Level Graph Theoretic Design and Development of Complex Information system", in *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing (ITCC'00)*, Las Vegas, NV, March 27-29, 2000.

N. Schneidewind, "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metric", in *IEEE Transactions on Software Engineering*, Vol. 25, No. 6, November/December 1999, pp. 769-781.

N. Schneidewind, "Cost Framework for COTS Evaluation", in *Proceedings of COMPSAC 99*, Phoenix, AZ, 27 October 1999, pp. 100-101.

N. Schneidewind and A. Nikora, "Predicting Deviations In Software Quality By Using Relative Critical Value Deviation Metrics", in *Proceedings of The 10th International Symposium on Software Reliability Engineering*, Boca Raton, Florida, November 1-4, 1999, pp. 136-146.

N. Schneidewind, "Investigation of the Risk to Software Reliability of Requirements Changes", in *Proceedings of the 1999 NASA Workshop on Risk Management*, Morgantown, West Virginia, October 28-29, 1999.

N. Schneidewind, "Reliability Modeling for Safety Critical Software", in *Proceedings of the 12th Annual Software Technology Conference*, Salt Lake City, Utah, 4/30-5/5, 2000.

N. Schneidewind, "Software Quality Control and Prediction Model for Maintenance", in *Annuals of Software Engineering*, Vol. 9, 2000, pp. 79-101.

N. Schneidewind, "The Ruthless Pursuit of the Truth about COTS", in *Proceedings of the North Atlantic Treaty Organization, Commercial Off-The-Shelf Products In Defense Applications*, Brussels, Belgium, 3-5 April 2000.

D. Zhang and Luqi, "Approximate Declarative Semantics for Rule Base Anomalies", *Knowledge-Based Systems,* Vol.12, No.7, November 1999, pp.341-353.

D. Zhang, "Applying Machine Learning Algorithms in Software Development", in *Proceedings of 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario"*, Santa Margherita Ligure, Italy, June 13-16, 2000. Also available on-line at http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/

Scientific Personnel:

Dr. Norman **Schneidewind**, Professor, NPS.

Dr. Du Zhang, Visiting Professor, NPS.

Dr. Swapan Bhattacharya (National Research Council Research Associate)

Dr. Jiang Guo (National Research Council Research Associate)

Dr. Jun Ge (National Research Council Research Associate)

Dr. Mikhail Auguston (National Research Council Research Associate)

Dr. Oleg Kiselyov (National Research Council Research Associate)

T. Augustine, "Naval Architecture Environment: Facilitating JV2010", Master's thesis, Software Engineering, NPS, December 1999.

K. Gee, "An Architectural Framework For Integrating Cots/Gots/Legacy Systems", Master's thesis, Software Engineering, NPS, June 1999.

T. Nguyen, "Commercial Off-The-Shelf (Cots)/Legacy Systems Integration Architectural Design And Analysis", Master's thesis, Software Engineering, NPS, September 2000.

T. Tran and J. Allen, "Interoperability And Security Support For Heterogeneous Cots/Gots/Legacy Component-Based Architecture", Master's thesis, Software Engineering, NPS, September 2000.

N. Cheng, "Automated generation of wrappers for interoperability", Master's thesis, Computer Science, NPS, June 2000.

Report of Inventions: N/A

Scientific Progress and Accomplishments:

This project addresses the problem of how to produce reliable software that is also flexible and cost effective for the DoD distributed software domain. Current and future DoD software systems fall into two categories: information systems and warfighter systems. Both kinds of systems can be distributed, heterogeneous and network-based, consisting of a set of components running on different platforms and working together via multiple communication links and protocols.

We focused on "wrap and glue" technology based on a domain specific distributed prototype model. Glue and wrappers consists of software that bridges the interoperability gap between individual COTS/GOTS components. The key to making the proposed approach reliable, flexible, and cost-effective is the automatic generation of glue and wrappers based on a designer's specification. The proposed "wrap and glue" approach allows system designers to concentrate on the difficult interoperability problems and defines solutions in terms of deeper and more difficult interoperability issues, while freeing designers from implementation details. The objective of our research is to develop an integrated set of formal models and methods for system engineering automation. These results will enable building decision support tools for concurrent engineering. Our research addresses complex modular systems with embedded control software and real-time requirements.

Our long-term goals are to construct an integrated set of software tools that can improve software quality and flexibility by automating a significant part of the process and providing substantial decision support for the aspects that cannot be automated. The resulting development environment should be adaptable to enable (1) maintaining integrated support in the presence of business process improvement, (2) incorporation of future improvements in engineering automation methods, and (3) specialization to particular problem domains.

Specific tasks accomplished in FY00 include (1) the design of an interface wrapper model that allows developers to treat distributed objects as local objects, (2) the development of a tool to generate Java interface wrappers from a specification written in the high-level Prototype System Description Language (PSDL), (3) the design of a distributed heterogeneous environment to automate the process of integration distributed systems, (4) a case study involving the development of a "wrapper and glue" solution for integrating/extending COTS/GOTS/legacy components of the Naval Integrated Tactical Environmental System I (NITES I), (5) the design of high-level net models for fault detection in multistage interconnected networks, (6) tools for assertion checking, dynamic analysis and testing of programs, (7) application of machine learning algorithms in software development, and (8) reliability modeling for safety critical software.

Technology Transfer:

M. Auguston, "Assertion Checker for the C Programming Language based on Computations over event traces", presented at the Fourth International Workshop on Algorithmic and Automatic Debugging, AADEBUG'2000, Munich, Germany, August 28-30, 2000.

M. Auguston, "Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars", presented at the 12th International Conference on Software Engineering and Knowledge Engineering, Chicago, USA, July 6-8, 2000.

M. Auguston, "Lightweight semantics models for program testing and debugging automation", presented at the 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000.

V. Berzins, "Prototyping Tool-Kit (APT)", presented at the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000), ICSE 2000, Limerick, Ireland, June 4-11, 2000.

V. Berzins, "Static Analysis for Program Generation Templates", presented at the 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000.

V. Berzins, member, Steering Committee, 2000 ARO/NSF/CNR Monterey Workshop On Modeling Software System Structures in a Fastly Moving Scenario, held in Santa Margherita Ligure, Italy, June 13-16, 2000.

S. Bhattacharya, "Automated Generation of Wrappers for Interoperability", presented at the 2000 Command and Control Research and Technology Symposium, Monterey, CA, June 26-28, 2000.

O. Kiselyou, "Implementing Metcast in Scheme", presented at the Workshop on Scheme and Functional Programming, Montréal, September, 2000, pp. 23-25.

Luqi, Chair of the Program Committee, the 11th IEEE International Workshop on Rapid System Prototyping, was held in Paris, France, June 21-23, 2000.

Luqi, Co-Chair, Program Committee, 2000 ARO/NSF/CNR Monterey Workshop On Modeling Software System Structures in a Fastly Moving Scenario, held in Santa Margherita Ligure, Italy, June 13-16, 2000.

M. Shing, "Computer Aided Prototyping System (CAPS) for Heterogeneous Systems Development and Integration", presented at the 2000 Command and Control Research and Technology Symposium, Monterey, CA, June 26-28, 2000.

N. Schneidewind, Panel Member, "Can Metrics and Models be Applied Across Multiple Releases or Projects?", presented at the Sixth International Metrics Symposium, Boca Raton, Florida, November 4-6, 1999.

N. Schneidewind, "Predicting Deviations In Software Quality By Using Relative Critical Value Deviation Metrics", presented at the Tenth International Symposium on Software Reliability Engineering, Boca Raton, Florida, November 1-4, 1999.

N. Schneidewind, Panel Member, "Practical Issues in Implementing Software Reliability Measurement", presented at the Tenth International Symposium on Software Reliability Engineering, Boca Raton, Florida, November 1-4, 1999.

N. Schneidewind, Keynote Talk, "Investigation of the Risk to Software Reliability of Requirements Changes", presented at the 1999 NASA Workshop on Risk Management, Morgantown, West Virginia, October 28-29, 1999.

N. Schneidewind, "Cost Framework for COTS Evaluation", presented at the COMPSAC 99, Phoenix, AZ, 27 October 1999.

N. Schneidewind and M. Sahinoglu, Tutorial, "New Advances in Software Reliability Modeling", presented at the Fifth Biennial World Conference on Integrated Design and Process Technology, Dallas Texas, June 6, 2000.

N. Schneidewind, "The Impact of Software Reliability, Dependability, and Security in the 21st Century", presented at the Fifth Biennial World Conference on Integrated Design and Process Technology, Dallas Texas, June 6, 2000.

N. Schneidewind, Tutorial, "A Roadmap To Distributed Client-Server Software Reliability Engineering", presented at the Quality Week 2000, San Francisco, California, May 30, 2000.

N. Schneidewind, Panel Chair, "Developing the Next Generation IEEE Dependability Standard", presented at the Software Technology Conference, Salt Lake City, Utah, May 4, 2000.

N. Schneidewind, "The Ruthless Pursuit of the Truth about COTS", presented at the North Atlantic Treaty Organization, Commercial Off-The-Shelf Products In Defense Applications, "The Ruthless Pursuit of COTS", Information Systems Technology Panel (IST), Brussels, Belgium, 3-5 April 2000.

D. Zhang, "Applying Machine Learning Algorithms in Software Development", presented at the 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000.

APPENDICES

Assertion checker for the C programming language based on computations over event traces

Mikhail Auguston

Computer Science Department, New Mexico State University
Las Cruces, NM 88003, USA
phone: (505) 646-5286, fax: (505) 646-1002
mikau@cs.nmsu.edu
http://www.cs.nmsu.edu/~mikau

ABSTRACT

This paper suggests an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. Assertions can capture the dynamic properties of a particular target program and can formalize the general knowledge of typical bugs and debugging strategies. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation.

An implementation architecture and preliminary experiments with a prototype assertion checker for the C programming language are discussed.

Keywords

Program behavior models, events, event grammars, software testing and debugging automation.

1 INTRODUCTION

Program testing and debugging is still a human activity performed largely without any adequate tools, and consuming more than 50% of the total program development time and effort [9]. Testing and debugging are mostly concerned with the program run-time behavior, and developing a precise *model of program behavior* becomes the first step towards any dynamic analysis automation. In building such a model several considerations were taken in account. The first assumption we make is that the model is discrete, i.e. comprises a finite number of well-separated elements. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) do not require the total ordering of actions, so *partial event ordering* is the most adequate method for this purpose [21].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship, events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

Finally, the program execution can be modeled as a set of events (*event trace*) with two basic relations: partial ordering and inclusion. In order to specify meaningful program behavior properties we have to enrich events with some attributes.

An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of the event), etc. This program behavior model may be regarded as a "lightweight" semantics of the programming language.

The next problem to be addressed after the program behavior model is set up is the formalism for specifying properties of the program behavior. This could be done in many different ways, e.g., by adopting some kind of logic calculi (predicate logic, temporal logic). Such a direction leads to tools for static program verification, or in more pragmatic incarnations to an approach called model checking [12].

Since our goal is dynamic program analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we developed the

concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN ([3], [17]) based on a functional paradigm and the use of event patterns and aggregate operations over events. The papers [2], [3], [17] are based on our assertion checker prototype for a subset of the PASCAL language. This paper describes the first experience with an assertion checker for the C programming language. The implementation of the C assertion checker is based on source code automatic instrumentation and supports almost complete C language (the most serious constraint is the requirement that the target program is contained in a single compilation unit). To adjust to the specifics of the C target language the FORMAN language has been modified, in particular, the scope construct (WITHIN function-name) and explicit type cast have been added (see examples in Sec. 4).

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data flow and control flow in the target program. Assertions can also be used as conditions in rules which describe debugging actions. For example, an error message is a typical action for a debugger or consistency checker. Thus, it is also possible to specify debugging strategies.

The notions of event and event type are powerful abstractions which make it possible to write assertions independent of a particular target program. Such generic assertions can be collected in standard libraries which represent general knowledge about typical bugs and debugging strategies and could be designed and distributed as special software tools.

Possible applications of a language for computations over a program event trace include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [5]. Even the traditional debugging method based on scattering print statements across the source code may be easily implemented as an appropriate computation on the event trace (see example in Sec 4). The advantage is that the print statements are kept in a separate file and the source code of the target program will be instrumented automatically just before execution. A study of applying FORMAN to parallel programming is presented in [4].

2 EVENTS

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a function is called, a statement is executed, or some expression is evaluated. A particular action may be performed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

In order to give some rationale for our notion of an event, let us consider a well-known idea such as a counter. Usually the history of a variable X when used as a counter looks like:

> X := 0; ...
>
> Loop ...
>
> > X := X + 1; ...
>
> endloop; ...

In order to determine whether the actual behavior of the counter X matches the pattern described by the program fragment above we have to consider the following events. Let Initialize_X denotes the event of assigning 0 to the variable X, Augment_X denotes the event of incrementing X, and Assign_X denotes the event of assigning any value to the variable X. The event Assign_X is a composite one; it contains either Initialize_X or Augment_X events. One could determine if X behaves as a counter when a program segment S is executed in the following way. First, the sequence A of all events of the type Assign_X from the event trace of program segment S has to be extracted preserving the ordering between events. Second, A has to be matched with the pattern:

```
Initialize_X (Augment_X) *
```

where '*' denotes repetition zero or more times. If the actual sequence of events does not match this pattern we can report an error. Therefore, assertion checking can be represented as a kind of computation over a target program event trace.

The program state (current values of variables) can be considered at the beginning or at the end of an appropriate event. This provides the opportunity to write assertions about program variable values at different points in the program execution history.

Program profiling usually is based on counting the number of events of some type, e.g. the number of statement executions or procedure calls. Performance measurements may be based on attaching the duration attribute to such

events and summarizing durations of selected events.

3 PROGRAM BEHAVIOR MODEL

FORMAN is intended to be used to specify behavior of programs written in some high-level programming language which is called the *target language*. The model of target program behavior is formally defined as a set of events (*event trace*) with two basic relations, which may or may not hold between two arbitrary events. The events may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more than one of these relations can be established.

In order to define the behavior model for a particular target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (e.g., rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object, in which case the grammar describes how the event is split into other event sequences or sets. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different types in the program execution history; it is not intended to be used for parsing an actual event trace.

The rule A :: B C establishes that if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C also exist, such that the relations b IN a, c IN a, b PRECEDES c hold.

For the C language assertion checker prototype we have defined the following simple event grammar.

(Axiom 1)   execute_program::

    (ex_stmt | eval_expr)*

(Axiom 2)   ex_stmt::

    (ex_stmt | eval_expr)*

(Axiom 3) eval_expr:: func_call |

    eval_expr+ destination? |

    { eval_expr } +

(Axiom 4) func_call::

    { eval_expr }* ex_stmt*

Axiom 1 states that the program execution event contains (the IN relation) a set of zero or more ordered (w.r.t. relation PRECEDES) events of the types execute-statement or evaluate-expression.

Axiom 2 states the same fact about the execute_statement event. For example, the event of executing a composite statement such as if-then-else will contain an event

eval_expr for condition evaluation and a sequence of zero or more events for the corresponding THEN or ELSE branch execution. If a statement has a label attached, the label traversal itself is considered as an empty statement execution event.

Axiom 3 describes the possible structure of an expression evaluation event: it may contain a function call event or may be an ordered sequence of other expression evaluation events (e.g. for a 'comma" expression). The assignment expression evaluation contains the event destination which is distinguished because it is of a special importance for assertion checking. In our model we have avoided any assumptions about the ordering of argument evaluation for binary operations, such as '+' or '*', since the C language semantics leaves this undefined [18]. The metaexpression {eval_expr}+ denotes a set of one or more events of the type eval_expr without any ordering relationship.

Axiom 4 describes the structure of a function call event which starts with a set (may be empty) of unordered events for actual parameter evaluation followed by the function body execution events.

The order of event occurrences reflects the semantics of the target language. When performing an assignment statement, first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, such as the source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, the duration of the event, a previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of the target language to write assertions about target program variables.

Events can be described by patterns which capture the structure of event and context conditions. Program paths can be described by regular path expressions over events.

4 EXAMPLES OF DEBUGGING RULES

In general, a *debugging rule* performs some actions that may include computations over the target program event trace. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog to the user as well. An assertion is a boolean expression that may

10

contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or violated. A debugging rule has the form:

```
assertion   SAY (expression sequence)
            ONFAIL SAY (expression sequence)
```

The presence of metavariables in the assertion makes it possible to use FORMAN as a debugger's query language. The evaluation of an assertion is interrupted when it becomes clear that the final value will be False (or True), and the current values of metavariables can be used to generate readable and informative messages.

We will use as an example of a C program the Simple Tokenizer program described in [25]. This program reads a text file until the special symbol '.' (dot) is read, recognizes small integers, identifiers, and some predefined key words, skips spaces and PASCAL-like comments, prints the input text with line numbers attached before each line, splits the output into pages with a page header on the top of each page (including page number), and reports each token recognized. Unrecognized symbols are printed as ERROR tokens. The source code contains 542 lines of code (including some of our updates and comments). The following list of function prototypes used in the Simple Tokenizer gives some idea of the architecture.

```
void init_scanner(char *name);
void init_page_header(char *name);
BOOLEAN get_source_line();
void get_char();
void skip_blanks();
void skip_comment();
void get_token();
void get_word();
BOOLEAN is_reserved_word();
void get_number();
void get_special();
void open_source_file(char *name);
void close_source_file();
void print_line(char line[]);
void print_token();
void print_page_header();
void quit_scanner();
```

The input text file for Simple Tokenizer used for running the following examples contained 150 lines of text with a total of 454 tokens. The corresponding output contained 13 pages with maximum of 50 lines per page (including the input lines and messages about tokens recognized, each on a separate line of output).

Example of a debugging query.

In order to obtain the history of a global variable `page_number` the following computation over the event trace can be performed. The WITHIN construct indicates the scope of the trace computations defined by this rule. The rule condition is TRUE, and as a side effect the entire history of variable `page_number` is shown. The [...] list constructor defines a loop over the entire program event trace (`execute_program` event). All events matching the pattern `func_call IS printf` (i.e. events of the type func_call and function name 'printf') executed within the body of `print_page_header` function are selected from the trace and the function VALUE is applied to them. The metavariable C holds the event `func_call` under consideration. The resulting sequence consists of variable `page_number` values at the end of each event captured by metavariable C during the program execution.

```
WITHIN print_page_header
     TRUE
SAY( 'The history of page_number variable
values is: '
 [ C: func_call IS 'printf'
      FROM execute_program
      APPLY VALUE(int)(AT C page_number) ]);
END
```

When executed on our prototype the following output is produced:

```
The history of page_number variable values
is: 1 2 3 4 5 6 7 8 9 10 11 12 13
```

This debugging rule provides a slice of the program execution history containing the trace of particular variable values. The matter of interest may be, for instance, to check whether the values in the variable history are arranged in ascending order.

Example of an assertion checking.

Let us write and check the assertion: *"There exists an input line with length exceeding some maximum, say 10."* The program snippet containing the function get_source_line

11

looks like:

```
BOOLEAN     get_source_line()
{char
print_buffer[MAX_SOURCE_LINE_LENGTH+9];
    if((fgets(source_buffer,
            MAX_SOURCE_LINE_LENGTH,
            source_file)) != NULL) {
    ++line_number;
Get_Line:
    sprintf(print_buffer, "%4d %d: %s",
    line_number,level,source_buffer);
    print_line(print_buffer);
    return(TRUE);
    }
    else return(FALSE);            }
```

Traversal of a label is an event of the type ex_stmt, and we can check the value of a C expression strlen(source_buffer) > 10 after this event.

```
WITHIN get_source_line
    EXISTS L: ex_stmt IS 'Get_Line:'
            FROM execute_program
VALUE(int)(AT L strlen(source_buffer) >10)
SAY('Too long input line detected at stmt' )
SAY(L)
SAY( 'It is '
    VALUE(int)(AT L strlen(source_buffer))
    'characters long')
ONFAIL SAY(' No long input lines detected');
```

We check whether the expression strlen(source_buffer) > 10 is not equal to 0 for all events L. When the assertion is satisfied for the first time, the assertion evaluation terminates and the current value of the metavariable L can be used for message output. In order to make error messages more informative, the value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text
source_line_number within function_name
Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of the step counter.

When executed on our prototype this assertion checking

yields the following output.

```
Too long input line detected at stmt
ex_stmt :> 'Get_Line:' source line 460
within function get_source_line
Time= 95 .. 96
It is 20 characters long
```

Example of a run time statistics gathering.

It is hard to measure real execution time of a heavily instrumented target program, although the simulated time measurement may be performed given that events may have some duration attributes predefined. In order to obtain the actual number of function calls executed, number of function get_source_line calls, and number of tokens recognized by the Simple Tokenizer, the following query can be performed:

```
TRUE
SAY('Total function calls'
    CARD[ ALL func_call
            FROM execute_program])
SAY('Total function get_source_line calls'
    CARD [ func_call IS get_source_line
            FROM execute_program])
SAY('Total tokens recognized'
    CARD [ ALL func_call IS get_token
            FROM execute_program]
    ', among them '
    CARD [ ALL F: func_call &
    SOURCE_TEXT(F) == 'get_token'
    AND VALUE (int)(AT F token == ERROR)
            FROM execute_program]
    'ERROR tokens detected' );
```

The CARD operator returns the number of items selected by the aggregate operation, i.e. the number of events matching the pattern in the aggregate operation body. The ALL option in the aggregate operation indicates that all nested events of the type func_call should be taken into account. The pattern in the third aggregate operation provides an example of a complex event pattern with a context condition attached. The scope of this trace computation is the entire program trace. After execution on our prototype the

following output is obtained.

```
Total function calls 6802
Total function get_source_line calls 150
Total tokens recognized 454, among them 37
    ERROR tokens detected
```

Example of path expression checking.

Regular expressions over event patterns may describe sequences of events extracted from the event trace. The following assertion checks whether function get_token and print_token calls appear in a certain order. Sequence of events satisfying the pattern X:func_call& SOURCE_TEXT(X) == 'get_token' OR SOURCE_TEXT(X)=='print_token' is selected from the entire event trace and matched against the path expression (func_call IS 'get_token' func_call IS 'print_token') +. A message is produced with information about the pattern matching results.

```
[ X: func_call & SOURCE_TEXT(X)==

            'get_token' OR

                    SOURCE_TEXT(X)==
'print_token' FROM execute_program ]
        SATISFIES(func_call IS 'get_token'

        func_call IS 'print_token' ) +
    SAY('function calls follow the pattern

        (get_token print_token) + ')
    ONFAIL SAY( 'pattern

        (get_token print_token) +

        is violated');
```

Example of instrumenting the target source code with print statements.

Suppose we want to insert in the target source code print statements to print at run time the value of input strings with length exceeding 10 and corresponding line numbers. Values of interest are available in global variables source_buffer and line_number, respectively. The following debugging rule performs this function.

```
WITHIN get_source_line
FOREACH L1: ex_stmt IS 'Get_Line:'
            FROM execute_program
    VALUE ( int )
(  AT  L1  strlen(source_buffer)>10?
```

```
printf("long line!!!\n%s\n",source_buffer):1)
    AND

        VALUE ( int )
    ( AT L1
    printf("line_number=%d\n",line_number));
    END
```

Formally this rule will cause an assertion checking, which will be successful since the C expression involved yields a non-zero value (representing Boolean TRUE); as a side effect the print statements are executed at run time. This debugging rule has two aspects worthy of notice. First, the instrumentation code is separated from the target code; it will be inserted automatically just before the execution and can be maintained in a separate file. There may be several different print instrumentations defined for the same target program; keeping them in separate files provides a great flexibility in arranging a custom set of print statements to be inserted at run time. Second, the instrumentation is attached to a particular event in the trace matching the pattern ex_stmt IS 'Get_Line:', i.e. traversal of the label Get_Line:, therefore it does not depend on possible target code modifications as long as the label is not changed.

Debugging rules can be considered as a way of formalizing reasoning about the target program execution -- humans often use similar patterns for reasoning when debugging programs. For example, if the index expression of an array element is out of range, the debugger can try a rule for eval-index events that invokes another rule about a wrong value of the event eval-expression, which in turn will cause investigation of histories of all variables included in the expression.

5 BRIEF IMPLEMENTATION SURVEY

The architecture of the computations over the event traces for the C programming language is based on the automatic instrumentation of the target program source code in such a way that some computations over the trace are performed at run time and the rest of information is saved in the trace file for postmortem processing. The instrumentation does not change the semantics of the target program. The trace file is read by the FORMAN interpreter to complete the computations over the trace and to generate messages. A special attempt in this prototype was made to optimize the trace generation, in particular to filter events in order to reduce the size the trace.

The front end of the assertion checker was adapted and modified from Shawn's Flisakowski parser and abstract syntax tree builder for the complete C programming language (gcc version) [14]. The instrumentation module was designed by Ana Erendira Flores-Mendoza as her Master's project in the NMSU CS Department [15]. The total size of the software used for the prototype amounts to more then 20KLOC of C/

13

lex/yacc/Rigal [1] code.

Since an event in our model has a duration and may contain another events, it is represented on the trace by two records, one for the beginning of event and one for the end. The semantics of the C language do not specify the order of subexpression execution; to address this issue and to ensure proper nesting of event eval_expr beginning and end records on the trace the instrumented code maintains some auxiliary stack for expression evaluation. A similar stack mechanism is added to the instrumented code to maintain proper nesting of ex_stmt and func_call events when performing return, goto, and break statements. These specifics of our target program behavior model led as to the decision to implement the instrumentation module from the scratch rather than to use some generic instrumentation tools like [33]. The basic building block for expression E instrumentation is comma-expression (e1, temp = E, e2, temp), where e1 stands for prologue instrumentation, e2 stands for epilog instrumentation, and temp variable holds the result of the original expression E evaluation.

Only events necessary for the given FORMAN program are involved in the computations over the trace and put on the trace. For the Simple Tokenizer program discussed above, using the input file with 150 lines and 454 tokens and the entire set of debugging rules described in the previous section the total number of events generated by the target program according to the event grammar is 105,808, although only 7253 of them (less then 7%) are put on the trace. Even in its current state with many potential optimizations not yet implemented, the prototype demonstrates the feasibility of trace computations for "typical" student programs like the Simple Tokenizer. Our experiments with other C programs show that storing several tens of thousands of events on the trace is sufficient for a large number of "typical" C programs run with a set of debugging rules and assertions similar to the examples in Sec. 4. It should be noted that typically the size of input data used for testing and debugging purposes is relatively small.

6 RELATED WORK

What follows is a very brief survey of basic ideas known in Debugging Automation to provide the background for the approach advocated in this paper.

Event Notion

The Event Based Behavioral Abstraction (EBBA) method suggested in [7] characterizes the behavior of the entire program in terms of both primitive and composite events. Context conditions involving event attribute values can be used to distinguish events. EBBA defines two higher-level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering

serves to eliminate from consideration events which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

An event-based debugger for the C programming language called Dalek [27] provides a means for describing user-defined events which typically are points within a program execution trace. A target program has to be instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events.

FORMAN has a more comprehensive modeling approach than EBBA or Dalek, based on the event grammar. A language for expressing computations over execution histories is provided, which is missing in EBBA and Dalek. The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events. FORMAN supports the design of universal assertions and debugging rules that could be used for debugging of arbitrary target programs. This generality is missing in the EBBA and Dalek approaches. The event in FORMAN is a time interval, in contrast with the event notion in previous approaches where events are considered pointwise time moments.

The COCA debugger [13] for the C language uses the GDB debugger for tracing and PROLOG for debugging queries execution. It provides a certain event grammar for C traces and event patterns based on attributes for event search. The query language is designed around special primitives built into the PROLOG query evaluator. We assume that FORMAN is more suitable for trace computations as it has been designed for this specific purpose.

Path Expressions

Data and control flow descriptions of the target program are essential for testing and debugging purposes. It is useful to give such a description in an explicit and precise form. The path expression technique introduced for specifying parallel programs in [11] is one such formalism. Trace specifications also are used in [26] for software specification. This technique has been used in several projects as a background for high-level debugging tools, (e.g. in [10]), where path rules are suggested as a kind of debugger commands. FORMAN provides a flexible language means for trace specification including event patterns and regular expressions over them.

Assertion Languages

Assertion (or annotation) languages provide yet another approach to debugging automation. The approaches currently in use are mostly based on boolean expressions attached to selected points of the target program, like the assert macro in C [18]. The ANNA [23] annotation language for the Ada target language supports assertions on variable and type declarations. In the TSL [22], [29] annotation language for Ada the notion of event is introduced in order to describe the

14

behavior of Tasks. Patterns can be written which involve parameter values of Task entry calls. Assertions are written in Ada itself, using a number of special pre-defined predicates. Assertion-checking is dynamic at run-time, and does not need post-mortem analysis. The RAPIDE project [24] provides an event-based assertion language for software architecture description.

In [6] events are introduced to describe process communication, termination, and connection and detachment of process to channels. A language of Behavior Expressions (BE) is provided to write assertions about sequences of process interactions. BE is able to describe allowed sequences of events as well as some predicates defined on the values of the variables of processes. Event types are process communication and interactions such as send, receive, terminate, connect, detach. Evaluation of assertions is done at run-time. No composite events are provided.

Another experimental debugging tool is based on trace analysis with respect to assertions in temporal interval logic. This work is presented in [20] where four types of events are introduced: assignment to variables, reaching a label, interprocess communication and process instantiation or termination. Composite events cannot be defined. Different varieties of temporal logic languages are used for program static analysis called Model Checking [12].

In [30] a practical approach to programming with assertions for the C language is advocated, and it is demonstrated that even local assertions associated with particular points within the program may be extremely useful for program debugging.

The DUEL [19] debugging language introduces expressions for C aggregate data exploration, for both assertions and queries.

The FORMAN language for computations over traces provides a flexible means for writing both local and global assertions, including those about temporal relations between events.

Algorithmic Debugging

The original algorithmic program debugging method was introduced in [32] for the Prolog language. In [31] and [16] this paradigm is applied to a subset of PASCAL. The debugger executes the program and builds a trace execution tree at the procedure level while saving some useful trace information such as procedure names and input/output parameter values. The algorithmic debugger traverses the execution tree and interacts with the user by asking about the intended behavior of each procedure. The user has the possibility to answer "yes" or "no" about the intended behavior of the procedure. The search finally ends and a bug is localized within a procedure p when one of the following holds: procedure p contains no procedure calls, or all procedure calls performed from the body of procedure p

fulfill the user's expectations.

Algorithmic debugging can be considered as an example of debugging strategy, based on some assertion language (in this case assertions about results of a procedure call). The notion of computation over execution trace introduced in FORMAN may be a convenient basis for describing such debugging strategies.

7 CONCLUSIONS

In brief, our approach can be explained as "computations over a target program event trace based on a precise program behavior model". According to [8] and [28], approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e., bugs which could be detected by appropriate assertion checking similar to that demonstrated above.

We expect the advantages of our approach to be the following:

- The notion of **an event grammar** provides a general basis for program behavior models. In contrast with previous approaches, the **event** is not a point in the trace but an interval with a beginning and an end.

- Event grammar provides a coordinate system to refer to any interesting event in the execution history. Event attributes provide complete **access to each target program's execution state**. Assertions about particular execution states as well as assertions about sets of different execution states may be checked.

- The IN relation yields a **hierarchy of events**, so the assertions can be defined at an appropriate level of granularity.

- A language for **computations over event traces** provides a **uniform framework** for assertion checking, profiles, debugging queries, and performance measurements.

- The fact that assertions and other computations over the target program event trace can be **separated from the text of the target program** allows accumulation of formalized knowledge about particular programs and makes it easy to control the number of assertions to be checked.

The first experiments with our C assertion checker prototype prove that:

- instrumentation of the C source code may be an appropriate technique for automatic testing and debugging tool design,

- event filtering can reduce the size of the stored event trace to 5-20% of the total trace,

- the size of the stored event trace could be kept within reasonable limits (several tens of thousands of events) for realistic C programs.

The future work will be dedicated to further optimizations of trace computation and event filtering, and to the design of

an appropriate user interface.

References

[1] M. Auguston, Programming language RIGAL as a compiler writing tool, ACM SIGPLAN Notices, December 1990, vol.25, #12, pp.61-69

[2] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering,* Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.

[3] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", *in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging,* Saint-Malo, France, May 1995.

[4] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering,* vol.6, No 4, 1996, pp. 609-640.

[5] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering,* SEKE'97, Madrid, Spain, June 1997, pp. 257-262

[6] F. Baiardi, N. De Francesco, G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering,* vol. SE-12, No. 4, April 1986, pp. 547-553.

[7] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *The Journal of Systems and Software 3,* 1983, pp. 255-264.

[8] B. Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[9] F. Brooks, The Mythical Man-Month, 2nd edition, Addison-Wesley, 1995.

[10] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.

[11] R. H. Campbell, A. N. Habermann, "The specification of process synchronization by path expressions", *Lecture Notes in Computer Science,* vol. 16, 1974, pp. 89-102.

[12] E.Clarke et al., "Verification tools for Finite State Concurrent Systems", LNCS vol.803, 1994, pp.124-175.

[13] M.Ducasse, "COCA: An automated debugger for C", in Proceedings of the 1999 International Conference on Software Engineering, ICSE 99, Los Angeles, 1999, pp.504-513.

[14] Shawn Flisakowski, Parser and Abstract Syntax Tree Builder for the C Programming Language, ftp site at: ftp.cs.wisc.edu:/coral/tmp/spf/ctree-0.02.tar.gz

[15] Ana Erendira Flores-Mendoza, "C program instrumentation architecture for event trace collection", M.Sc. Thesis, Computer Science Department, New Mexico State University, Summer 1997.

[16] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing", *ACM LOPLAS -- Letters of Programming Languages and Systems.* Vol. 1, No. 4, December 1992.

[17] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software* 25, 1994, pp. 223-239.

[18] S.Harbison, G. Steele, "C: A Reference Manual", Fourth Edition, Prentice Hall, 1995.

[19] M.Golan, D.Hanson, "DUEL - A Very High-Level Debugging Language", in Proceedings of the Winter USENIX Technical Conference, San Diego, Jan. 1993.

[20] G. Goldszmidt, S. Katz, S. Yemini, "Interactive Blackbox Debugging for Concurrent Languages", *SIGPLAN Notices* vol. 24, No. 1, 1989, pp. 271-282.

[21] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM,* vol. 21, No. 7, July 1978, pp. 558-565.

[22] D. C. Luckham, D. Bryan, W. Mann, S. Meldal, D. P. Helmbold, "An Introduction to Task Sequencing Language, TSL version 1.5" (Preliminary version), Stanford University, February 1, 1990, pp. 1-68.

[23] D. C. Luckham, S. Sankar, S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications", *IEEE Software,* January 1991, pp.74-84.

[24] D. Luckham, J. Vera, "An Event-Based Architecture

Definition Language", *IEEE Transactions on Software Engineering, Vol.21, No. 9, 1995, pp. 717-734.*

[25] R.Mak, "Writing Compilers and Interpreters", John Wiley & Sons, 1991.

[26] J. McLean, "A Formal Method for the Abstract Specification of Software", *Journal of the Association of Computing Machinery*, vol.31, No. 3, July 1984, pp. 600-627.

[27] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", *Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.*

[28] S. L. Pfleeger, Software Engineering, Theory and Practice, Prentice Hall, 1998.

[29] D. Rosenblum, "Specifying Concurrent Systems with TSL", *IEEE Software*, May 1991, pp.52-61.

[30] D. Rosenblum, "A Practical Approach to Programming With Assertions", *IEEE Transactions on Software Engineering, Vol. 21, No 1, January 1995, pp. 19-31.*

[31] N. Shahmehri, "Generalized Algorithmic Debugging", Ph.D. Thesis No. 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1991.

[32] E. Shapiro, "Algorithmic Program Debugging", MIT Press, May 1982.

[33] K.Templer, C.Jeffrey, "A configurable automatic instrumentation tool for ANSI C", In the Proceedings of Automated Software Engineering Conference, 1998

17

Tools for program dynamic analysis, testing, and debugging based on event grammars

Mikhail Auguston
Computer Science Department, New Mexico State University
Las Cruces, NM 88003, USA
phone: (505) 646-5286, fax: (505) 646-1002
mikau@cs.nmsu.edu

ABSTRACT

This paper suggests an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation. Preliminary experiments with a prototype assertion checker for the C programming language are discussed.

1 INTRODUCTION

Program testing and debugging is still a human activity performed largely without any adequate tools, and consuming more than 50% of the total program development time and effort [8]. Testing and debugging are mostly concerned with the program run-time behavior, and developing a precise *model of program behavior* becomes the first step towards any dynamic analysis automation. In building such a model several considerations were taken in account. The first assumption we make is that the model is discrete, i.e. comprises a finite number of well-separated elements. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) do not require the total ordering of actions, so *partial event ordering* is the most adequate for this purpose [19].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship, events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of the event), etc. This program behavior model may be regarded as a "lightweight" semantics of the programming language.

The next problem to be addressed after the program behavior model is set up is the formalism for specifying properties of the program behavior. This could be done in many different ways, e.g., by adopting some kind of logic calculi (predicate logic, temporal logic). Such a direction leads to tools for static program verification, such as an approach called model checking [11].

Since our goal is dynamic program analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we developed the concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN ([2], [16]) based on a functional paradigm and the use of event patterns and aggregate operations over events. The papers [1], [2], [16] are based on our assertion checker prototype for a subset of the PASCAL language. This paper describes the first experience with an assertion checker for the C programming language. The implementation of the C assertion checker is based on source code automatic instrumentation. To adjust to the specifics of the C target language the FORMAN language has been modified, in particular, the scope construct (WITHIN function-name) and

18

explicit type cast have been added (see examples in Sec. 4).

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data flow and control flow in the target program.

Possible applications of a language for computations over a program event trace include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [4]. Even the traditional debugging method based on scattering print statements across the source code may be easily implemented as an appropriate computation on the event trace (see example in Sec 4). The advantage is that the print statements are kept in a separate file and the source code of the target program will be instrumented automatically just before execution. A study of applying FORMAN to parallel programming is presented in [3].

2 EVENTS

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a function is called, a statement is executed, or some expression is evaluated. A particular action may be performed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

In order to give some rationale for our notion of an event, let us consider a well-known idea such as a counter. Usually the history of a variable X when used as a counter looks like:

```
X := 0; ...
Loop ...
      X := X + 1; ...
endloop; ...
```

In order to determine whether the actual behavior of the counter X matches the pattern described by the program fragment above we have to consider the following events. Let Initialize_X denotes the event of assigning 0 to the variable X, Augment_X denotes the event of incrementing X, and Assign_X denotes the event of assigning any value to the variable X. The event Assign_X is a composite one; it contains either Initialize_X or Augment_X events. One could determine if X behaves as a counter when a program segment S is executed in the following way. First, the sequence A of all events of the type Assign_X from the event trace of program segment S has to be extracted preserving the ordering between events. Second, A has to be matched with the pattern:

```
Initialize_X (Augment_X)*
```

where '*' denotes repetition zero or more times. If the actual sequence of events does not match this pattern we can report an error. Therefore, assertion checking can be represented as a kind of computation over a target program event trace.

The program state (current values of variables) can be considered at the beginning or at the end of an appropriate event. This provides the opportunity to write assertions about program variable values at different points in the program execution history.

3 PROGRAM BEHAVIOR MODEL

FORMAN is intended to be used to specify behavior of programs written in some high-level programming language which is called the *target language*. The model of target program behavior is formally defined as a set of events (*event trace*) with two basic relations, which may or may not hold between two arbitrary events. The events may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more than one of these relations can be established.

In order to define the behavior model for a particular target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (e.g., rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object, in which case the grammar describes how the event is split into other event sequences or sets. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different types in the program execution history; it is not intended to be used for parsing an actual event trace.

The rule A :: B C establishes that if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C also exist, such that the relations b IN a, c IN a, b PRECEDES c hold.

For the C language assertion checker prototype we have

defined the following simple event grammar.

```
(Axiom 1)  execute_program::
              ( ex_stmt | eval_expr )*
(Axiom 2)  ex_stmt::
              ( ex_stmt | eval_expr )*
(Axiom 3)  eval_expr:: func_call |
              eval_expr+ destination? |
              { eval_expr } +
(Axiom 4)  func_call::
            { eval_expr }* ex_stmt*
```

Axiom 1 states that the program execution event contains (the IN relation) a set of zero or more ordered (w.r.t. relation PRECEDES) events of the types execute-statement or evaluate-expression.

Axiom 2 states the same fact about the execute_statement event. For example, the event of executing a composite statement such as if-then-else will contain an event eval_expr for condition evaluation and a sequence of zero or more events for the corresponding THEN or ELSE branch execution. If a statement has a label attached, the label traversal itself is considered as an empty statement execution event.

Axiom 3 describes the possible structure of an expression evaluation event: it may contain a function call event or may be an ordered sequence of other expression evaluation events (e.g. for a 'comma" expression). The assignment expression evaluation contains the event destination which is distinguished because it is of a special importance for assertion checking. In our implementation we have avoided any assumptions about the ordering of argument evaluation for binary operations, such as '+' or '*', since the C language semantics leaves this undefined [17]. The grammar rule {eval_expr}+ denotes a set of one or more events of the type eval_expr without any ordering relationship.

Axiom 4 describes the structure of a function call event which starts with a set (may be empty) of unordered events for actual parameter evaluation followed by the function body execution events.

The order of event occurrences reflects the semantics of the target language. When performing an assignment statement, first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, such as the source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, the duration of the event, a previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of the target language to write assertions about target program variables.

Events can be described by patterns which capture the structure of event and context conditions. Program paths can be described by regular path expressions over events.

4 EXAMPLES OF DEBUGGING RULES

In general, a *debugging rule* performs some actions that may include computations over the target program event trace. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. An assertion is a boolean expression that may contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or violated. A debugging rule has the form:

assertion   **SAY** (expression sequence)

ONFAIL SAY (expression sequence)

We will use as an example of a C program the Simple Tokenizer program described in [23]. This program reads a text file until the special symbol '.' (dot) is read, recognizes small integers, identifiers, and some predefined key words, skips spaces and PASCAL-like comments, prints the input text with line numbers attached before each line, splits the output into pages with a page header on the top of each page (including page number), and reports each token recognized. Unrecognized symbols are printed as ERROR tokens. The source code contains 542 lines of code (including some of our updates and comments). The input file used for running the following examples contained 150 lines of text with a total of 454 tokens. The corresponding output contained 13 pages with maximum of 50 lines per page (including the input lines and messages about tokens recognized, each on a separate line of output).

Example of a debugging query.
In order to obtain the history of a global variable page_number the following computation over the event trace can be performed. The WITHIN construct indicates the scope of the trace computations defined by this rule. The rule condition is TRUE, and as a side effect the entire history of variable page_number is shown. The [...] list constructor defines a loop over the entire program event trace (execute_program event). All events matching the pattern func_call IS printf executed within the body of print_page_header function are selected from the trace and the function VALUE is applied to them. The

20

metavariable C holds the event func_call under consideration. The resulting sequence consists of variable page_number values at the end of each event captured by metavariable C during the program execution.

```
WITHIN print_page_header

    TRUE

    SAY(   'The   history   of   page_number
variable values is: '

        [ C: func_call IS printf

            FROM execute_program

        APPLY VALUE(int)(AT C page_number) ]);

END
```

When executed on our prototype the following output is produced:

```
The history of page_number variable values
is: 1 2 3 4 5 6 7 8 9 10 11 12 13
```

This debugging rule provides a slice of the program execution history containing the trace of particular variable values. The matter of interest may be, for instance, to check whether the values in the variable history are arranged in ascending order.

Example of an assertion checking.

Let us write and check the assertion: *"There exists an input line with length exceeding some maximum, say 10."* The program snippet containing the function get_source_line looks like:

```
BOOLEAN     get_source_line()

{char   print_buffer[MAX_SOURCE_LINE_LENGTH  +9  ];

    if((fgets(source_buffer,

        MAX_SOURCE_LINE_LENGTH,

        source_file)) != NULL) {

    ++line_number;

    Get_Line:

    sprintf(print_buffer, "%4d %d: %s",

    line_number,level,source_buffer);

    print_line(print_buffer);

    return(TRUE);

    }

    else return(FALSE);                }
```

Traversal of a label is an event of the type ex_stmt, and we can check the value of a C expression strlen(source_buffer) > 10 after this event.

```
    WITHIN get_source_line

        EXISTS L: ex_stmt IS 'Get_Line:'

            FROM execute_program

    VALUE(int)(AT L strlen(source_buffer) >10)

    SAY('Too long input line detected at stmt' )

    SAY(L)

    SAY( 'It is '

        VALUE(int)(AT L strlen(source_buffer))

        'characters long')

    ONFAIL SAY(' No long input lines detected');
```

We check whether the expression strlen(source_buffer) > 10 is not equal to 0 for all events L. When the assertion is satisfied for the first time, the assertion evaluation terminates and the current value of the metavariable L can be used for message output. In order to make error messages more informative, the value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text

source_line_number

within function_name

Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of the step counter.

When executed on our prototype this assertion checking yields the following output.

```
Too long input line detected at stmt

ex_stmt  :> 'Get_Line:' source line 460
within function get_source_line

Time= 95 .. 96

It is 20 characters long
```

Example of a run time statistics gathering.

It is hard to measure real execution time of a heavily instrumented target program, although the simulated time measurement may be performed given that events may have some duration attributes predefined. In order to obtain the actual number of function calls executed, number of function get_source_line calls, and number of tokens recognized by the Simple Tokenizer, the following query can

be performed:

```
TRUE
SAY('Total function calls'
    CARD[ ALL func_call
          FROM execute_program])
SAY('Total function get_source_line calls'
    CARD [ func_call IS get_source_line
          FROM execute_program])
SAY('Total tokens recognized'
    CARD [ ALL func_call IS get_token
          FROM execute_program]
    ', among them '
    CARD [ ALL F: func_call &
     SOURCE_TEXT(F) == 'get_token'
     AND VALUE (int)(AT F token == ERROR)
               FROM execute_program]
    'ERROR tokens detected' );
```

The CARD operator returns the number of items selected by the aggregate operation, i.e. the number of events matching the pattern in the aggregate operation body. The ALL option in the aggregate operation indicates that all nested events of the type `func_call` should be taken into account. The pattern in the third aggregate operation provides an example of a complex event pattern with a context condition attached. The scope of this trace computation is the entire program trace. After execution on our prototype the following output is obtained.

```
Total function calls 6802
Total function get_source_line calls 150
Total tokens recognized 454 , among them 37
    ERROR tokens detected
```

Example of path expression checking.

Regular expressions over event patterns may describe sequences of events extracted from the event trace. The following assertion checks whether function `get_token` and `print_token` calls appear in a certain order. Sequence of events satisfying the pattern `X:func_call& SOURCE_TEXT(X) == 'get_token' OR SOURCE_TEXT(X) =='print_token'` is selected from the entire event trace and matched against the path expression `(func_call IS 'get_token' func_call IS 'print_token') +`. A message is produced with information about the pattern matching results.

```
[ X: func_call & SOURCE_TEXT(X) ==
          'get_token' OR
          SOURCE_TEXT(X) ==
```

```
    'print_token' FROM execute_program ]
        SATISFIES(func_call IS 'get_token'
        func_call IS 'print_token' ) +
    SAY('function calls follow the pattern
        (get_token print_token) + ')
ONFAIL SAY( 'pattern
        (get_token print_token) +
        is violated');
```

Example of instrumenting the target source code with print statements. Suppose we want to insert in the target source code print statements to print at run time the value of input strings with length exceeding 10 and corresponding line numbers. Values of interest are available in global variables `source_buffer` and `line_number`, respectively. The following debugging rule performs this function.

```
WITHIN get_source_line
FOREACH L1: ex_stmt IS 'Get_Line:'
            FROM execute_program
    VALUE ( int )
( AT L1 strlen(source_buffer)>10?
printf("long line!!!\n%s\n",source_buffer):1)
    AND
    VALUE ( int )
( AT L1
printf("line_number=%d\n",line_number));
END
```

Formally this rule will cause an assertion checking, which will be successful since the C expression involved yields a non-zero value (representing Boolean TRUE); as a side effect the print statements are executed at run time. This debugging rule has two aspects worthy of notice. First, the instrumentation code is separated from the target code; it will be inserted automatically just before the execution and can be maintained in a separate file. There may be several different print instrumentations defined for the same target program; keeping them in separate files provides a great flexibility in arranging a custom set of print statements to be inserted at run time. Second, the instrumentation is attached to a particular event in the trace matching the pattern `ex_stmt IS 'Get_Line:'`, i.e. traversal of the label `Get_Line:`, therefore it does not depend on possible target code modifications as long as the label is not changed.

5 BRIEF IMPLEMENTATION SURVEY

The architecture of the computations over the event traces for the C programming language is based on the automatic instrumentation of the target program source code in such a way that some computations over the trace are performed at

run time and the rest of information is saved in the trace file for postmortem processing. The instrumentation does not change the semantics of the target program. The trace file is read by the FORMAN interpreter to complete the computations over the trace and to generate messages. A special attempt in this prototype was made to optimize the trace generation, in particular to filter events in order to reduce the size the trace.

The front end of the assertion checker was adapted and modified from Shawn's Flisakowski parser and abstract syntax tree builder for the complete C programming language (gcc version) [13]. The instrumentation module was designed by Ana Erendira Flores-Mendoza as her Master's project in the NMSU CS Department [14]. The total size of the software used for the prototype amounts to more then 20KLOC of C/lex/yacc code.

Since an event in our model has a duration and may contain another events, it is represented on the trace by two records, one for the beginning of event and one for the end. The semantics of the C language do not specify the order of subexpression execution; to address this issue and to ensure proper nesting of event `eval_expr` beginning and end records on the trace the instrumented code maintains some auxiliary stack for expression evaluation. A similar stack mechanism is added to the instrumented code to maintain proper nesting of `ex_stmt` and `func_call` events when performing `return`, `goto`, and `break` statements. These specifics of our target program behavior model led as to the decision to implement the instrumentation module from the scratch rather than to use some generic instrumentation tools like [31].

Only events necessary for the given FORMAN program are involved in the computations over the trace and put on the trace. For the Simple Tokenizer example discussed above, using the input file with 150 lines and 454 tokens and the entire set of debugging rules described in the previous section the total number of events generated by the target program according to the event grammar is 105,808, although only 7253 of them (less then 7%) are put on the trace. Even in its current state with many potential optimizations not yet implemented, the prototype demonstrates the feasibility of trace computations for "typical" student programs like the Simple Tokenizer. Our experiments show that storing several tens of thousands of events on the trace is sufficient for "typical" C programs run with a set of debugging rules and assertions similar to the examples in Sec. 4. It should be noted that typically the size of input data used for testing and debugging purposes is relatively small.

6 RELATED WORK

What follows is a very brief survey of basic ideas known in Debugging Automation to provide the background for the approach advocated in this paper.

Event Notion

The Event Based Behavioral Abstraction (EBBA) method suggested in [6] characterizes the behavior of the entire program in terms of both primitive and composite events. Context conditions involving event attribute values can be used to distinguish events. EBBA defines two higher-level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering serves to eliminate from consideration events which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

An event-based debugger for the C programming language called Dalek [25] provides a means for describing user-defined events which typically are points within a program execution trace. A target program has to be instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events.

FORMAN has a more comprehensive modelling approach than EBBA or Dalek, based on the event grammar. A language for expressing computations over execution histories is provided, which is missing in EBBA and Dalek. The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events. FORMAN supports the design of universal assertions and debugging rules that could be used for debugging of arbitrary target programs. This generality is missing in the EBBA and Dalek approaches. The event in FORMAN is a time interval, in contrast with the event notion in previous approaches where events are considered pointwise time moments.

The COCA debugger [12] for the C language uses the GDB debugger for tracing and PROLOG for debugging queries execution. It provides a certain event grammar for C traces and event patterns based on attributes for event search. The query language is designed around special primitives built into the PROLOG query evaluator. We assume that FORMAN is more suitable for trace computations as it has been designed for this specific purpose.

Path Expressions

Data and control flow descriptions of the target program are essential for testing and debugging purposes. It is useful to give such a description in an explicit and precise form. The path expression technique introduced for specifying parallel programs in [10] is one such formalism. Trace specifications also are used in [24] for software specification. This technique has been used in several projects as a background for high-level debugging tools, (e.g. in [9]), where path rules are suggested as a kind of debugger commands. FORMAN provides a flexible language means for trace specification

including event patterns and regular expressions over them.

Assertion Languages

Assertion (or annotation) languages provide yet another approach to debugging automation. The approaches currently in use are mostly based on boolean expressions attached to selected points of the target program, like the assert macro in C [17]. The ANNA [21] annotation language for the Ada target language supports assertions on variable and type declarations. In the TSL [20], [27] annotation language for Ada the notion of event is introduced in order to describe the behavior of Tasks. Patterns can be written which involve parameter values of Task entry calls. Assertions are written in Ada itself, using a number of special pre-defined predicates. Assertion-checking is dynamic at run-time, and does not need post-mortem analysis. The RAPIDE project [22] provides an event-based assertion language for software architecture description.

In [5] events are introduced to describe process communication, termination, and connection and detachment of process to channels. A language of Behavior Expressions (BE) is provided to write assertions about sequences of process interactions. BE is able to describe allowed sequences of events as well as some predicates defined on the values of the variables of processes. Event types are process communication and interactions such as send, receive, terminate, connect, detach. Evaluation of assertions is done at run-time. No composite events are provided.

Another experimental debugging tool is based on trace analysis with respect to assertions in temporal interval logic. This work is presented in [18] where four types of events are introduced: assignment to variables, reaching a label, interprocess communication and process instantiation or termination. Composite events cannot be defined. Different varieties of temporal logic languages are used for program static analysis called Model Checking [11].

In [28] a practical approach to programming with assertions for the C language is advocated, and it is demonstrated that even local assertions associated with particular points within the program may be extremely useful for program debugging.

The FORMAN language for computations over traces provides a flexible means for writing both local and global assertions, including those about temporal relations between events.

Algorithmic Debugging

The original algorithmic program debugging method was introduced in [30] for the Prolog language. In [29] and [15] this paradigm is applied to a subset of PASCAL. The debugger executes the program and builds a trace execution tree at the procedure level while saving some useful trace information such as procedure names and input/output parameter values. The algorithmic debugger traverses the execution tree and interacts with the user by asking about the intended behavior of each procedure. The user has the possibility to answer "yes" or "no" about the intended behavior of the procedure. The search finally ends and a bug is localized within a procedure p when one of the following holds: procedure p contains no procedure calls, or all procedure calls performed from the body of procedure p fulfill the user's expectations.

Algorithmic debugging can be considered as an example of debugging strategy, based on some assertion language (in this case assertions about results of a procedure call). The notion of computation over execution trace introduced in FORMAN may be a convenient basis for describing such debugging strategies.

7 CONCLUSIONS

In brief, our approach can be explained as "computations over a target program event trace based on a precise program behavior model". According to [7] and [26], approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e., bugs which could be detected by appropriate assertion checking similar to that demonstrated above.

The first experiments with our C assertion checker prototype prove that:

- instrumentation of the C source code may be an appropriate technique for automatic testing and debugging tool design,
- event filtering can reduce the size of the stored event trace to 5-20% of the total trace,
- the size of the stored event trace could be kept within reasonable limits (several tens of thousands of events) for realistic C programs.

ACKNOWLEDGEMENTS

References

[1] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering,* Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.

[2] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", *in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging,* Saint-Malo, France, May 1995.

[3] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering,*

vol.6, No 4, 1996, pp. 609-640.

[4] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97,* Madrid, Spain, June 1997, pp. 257-262

[5] F. Baiardi, N. De Francesco, G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering,* vol. SE-12, No. 4, April 1986, pp. 547-553.

[6] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *The Journal of Systems and Software 3,* 1983, pp. 255-264.

[7] B. Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[8] F. Brooks, The Mythical Man-Month, 2nd edition, Addison-Wesley, 1995.

[9] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software 3,* 1983, pp. 265-276.

[10] R. H. Campbell, A. N. Habermann, "The specification of process synchronization by path expressions", *Lecture Notes in Computer Science,* vol. 16, 1974, pp. 89-102.

[11] E.Clarke et al., "Verification tools for Finite State Concurrent Systems", LNCS vol.803, 1994, pp.124-175.

[12] M.Ducasse, "COCA: A Debugger for C Based on Fine Grained Control Flow and Data Events", Tech. Report #1202, IRISA/ISNA, September 1998, pp.19

[13] Shawn Flisakowski, Parser and Abstract Syntax Tree Builder for the C Programming Language, ftp site at: ftp.cs.wisc.edu:/coral/tmp/spf/ctree-0.02.tar.gz

[14] Ana Erendira Flores-Mendoza, "C program instrumentation architecture for event trace collection", M.Sc. Thesis, Computer Science Department, New Mexico State University, Summer 1997.

[15] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing", *ACM LOPLAS -- Letters of Programming Languages and Systems.* Vol. 1, No. 4, December 1992.

[16] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software 25,* 1994, pp. 223-239.

[17] S.Harbison, G. Steele, "C: A Reference Manual", Fourth Edition, Prentice Hall, 1995.

[18] G. Goldszmidt, S. Katz, S. Yemini, "Interactive Blackbox Debugging for Concurrent Languages", *SIGPLAN Notices* vol. 24, No. 1, 1989, pp. 271-282.

[19] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM,* vol. 21, No. 7, July 1978, pp. 558-565.

[20] D. C. Luckham, D. Bryan, W. Mann, S. Meldal, D. P. Helmbold, "An Introduction to Task Sequencing Language, TSL version 1.5" (Preliminary version), Stanford University, February 1, 1990, pp. 1-68.

[21] D. C. Luckham, S. Sankar, S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications", *IEEE Software,* January 1991, pp.74-84.

[22] D. Luckham, J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering, Vol.21, No. 9, 1995, pp. 717-734.*

[23] R.Mak, "Writing Compilers and Interpreters", John Wiley & Sons, 1991.

[24] J. McLean, "A Formal Method for the Abstract Specification of Software", *Journal of the Association of Computing Machinery,* vol.31, No. 3, July 1984, pp. 600-627.

[25] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", *Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.*

[26] S. L. Pfleeger, Software Engineering, Theory and Practice, Prentice Hall, 1998.

[27] D. Rosenblum, "Specifying Concurrent Systems with TSL", *IEEE Software,* May 1991, pp.52-61.

[28] D. Rosenblum, "A Practical Approach to Programming With Assertions", *IEEE Transactions on Software Engineering, Vol. 21, No 1, January 1995, pp. 19-31.*

[29] N. Shahmehri, "Generalized Algorithmic Debugging", Ph.D. Thesis No. 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1991.

[30] E. Shapiro, "Algorithmic Program Debugging", MIT Press, May 1982.

[31] K.Templer, C.Jeffrey, "A configurable automatic instrumentation tool for ANSI C", In the Proceedings of Automated Software Engineering Conference, 1998

"Lightweight" Semantics Models for Program Testing and Debugging Automation

(Extended Abstract)

Mikhail Auguston
Computer Science Department, New Mexico State University,
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu
http://www.cs.nmsu.edu/~mikau

1 Introduction

We suggest an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. Assertions can capture both the dynamic properties of a particular target program and can formalize the general knowledge of typical bugs and debugging strategies. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation. Event grammars may be designed for sequential as well as for parallel programs. The approach suggested can be adjusted to a variety of programming languages. We illustrate these ideas on examples for the Occam and C programming languages.

Dynamic program analysis is one of the least understood activities in software development. A major problem is still the inability to express the mismatch between the expected and the observed behavior of the program on the level of abstraction maintained by the user [9]. In other words, a flexible and expressive specification formalism is needed to describe properties of the software system's implementation. Program testing and debugging is still a human activity performed largely without any adequate tools and consuming more than 50% of the total program development time and effort [8]. Debugging concurrent programs is even more difficult because of parallel activities, non-determinism and time-dependent behavior.

One way to improve the situation is to partially automate the debugging process. Precise *model of program behavior* becomes the first step towards debugging automation. It appears that traditional methods of programming language semantics definition don't address this aspect. In building such a model several considerations were taken in account. The first assumption we make is that the model is discrete, i.e. comprises a finite number of well-separated elements. This assumption is typical for Computer Science methods used for static and dynamic analysis of programs. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) don't require the total ordering of actions, so *partial event*

26

ordering is the most adequate method for this purpose [11].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

Finally, the program execution can be modeled as a set of events (*event trace*) with two basic relations: partial ordering and inclusion. The event trace actually is a model of program's behavior temporal aspect. In order to specify meaningful program behavior properties we have to enrich events with some attributes. An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of event), etc.

The next problem to be addressed after the program behavior model is set up is the formalism specifying properties of the program behavior. Since our goal is debugging automation, i.e. a kind of program dynamic analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we came up with the concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN [1], [3], [10] based on functional paradigm and the use of event patterns and aggregate operations over events.

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data and control flows in the target program. Assertions can also be used as conditions in rules which describe debugging actions. For example, an error message is a typical action for a debugger or consistency checker. Thus, it is also possible to specify debugging strategies.

The notions of event and event type are powerful abstractions which make it possible to write assertions independent of any target program. Such generic assertions can be collected in standard libraries which represent the general knowledge about typical bugs and debugging strategies and could be designed and distributed as special software tools.

FORMAN is a general language to describe computations over program event trace that can be considered as an example of *a special programming paradigm*. Possible application areas include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [5]. A study of FORMAN application for parallel programming is presented in [4]

2 Events, Event Traces, and the Language for Computations Over Event Traces

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a message is sent or received, a statement is executed, or some expression is evaluated. A particular action may be performed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

The model of target program behavior is formally defined through a set of general axioms about two basic relations, which may or may not hold between two arbitrary events: they may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more

than one of these relations can be established.

There are several general axioms that should be satisfied by any events a, b, c in the event trace of any target program.

1) Mutual exclusion of relations.

```
a PRECEDES b  => not (a IN b) and not (b IN a)
a IN b  =>  not(a PRECEDES b) and not (b PRECEDES a)
```

2) Noncommutativity.

```
a PRECEDES b  =>  not( b PRECEDES a)
a IN b  =>  not( b IN a)
```

3) Transitivity.

```
(a PRECEDES b ) and ( b PRECEDES c ) => ( a PRECEDES c)
(a IN b ) and ( b IN c ) => ( a IN c)
```

Irreflexivity for PRECEDES and IN follows from 2). Note that PRECEDES and IN are irreflexive partial orderings.

4) Distributivity

```
(a IN b) and (b PRECEDES c) => (a PRECEDES c)
(a PRECEDES b) and (c IN b) => (a PRECEDES c)
(FOR ALL a IN b (FOR ALL c IN d (a PRECEDES c) )) => (b PRECEDES d)
```

In order to define the behavior model for some target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object and the grammar describes how the event is split into other event sequences or sets. For example, the event execute-assignment-statement contains a sequence of events evaluate-right-hand-part and execute-destination. The evaluate-right-hand-part, in turn, consists of an unique event evaluate-expression. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different type in the program execution history, it is not intended to be used for parsing actual event trace.

The rule A :: (B C) establishes that if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C, also exist, such that the relations b IN a, c IN a, b PRECEDES c hold.

For example, the event grammar describing the semantics of a PASCAL subset may contain the following rules. The names, such as execute-program, and ex-stmt denote event types.

execute-program :: (ex-stmt *)

This means that each event of the type execute-program contains an ordered (w.r.t. relation PRECEDES)

sequence of zero or more events of the type ex-stmt.

> ex-stmt :: (label? (ex-assignment | ex-read-stmt | ex-write-stmt |
>
> ex-reset-stmt | ex-rewrite-stmt | ex-close-stmt | ex-cond-stmt |
>
> ex-loop-stmt | call-procedure))

The event of the type ex-stmt contains one of the events ex-assignment, ex-read-stmt, and so on. This inner event determines the particular type of statement executed and may be preceded by an optional event of the type label (traversing a label attached to the statement).

> ex-assignment :: (ex-righthand-part destination)

The order of event occurrences reflects the semantics of the target language. When performing assignment statement first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, for instance, source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, duration of the event, previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of target language to write assertions on target program variables [2] [3]. Events can be described by patterns which capture the structure of event and context conditions. Program paths can be described by regular path expressions over events.

The main extension for the parallel case [4] consists of the introduction of a new kind of composite event -- "snapshot," which can be considered an abstraction for the notion "a set of events that may happen at the same time." The "snapshot" event is a set of events each pair of which is not under the relation PRECEDES, this makes it possible to describe and to detect at run-time such typical parallel processing faults as data races and deadlock states.

3 Examples of Debugging Rules and Queries

In general, a *debugging rule* performs some actions that may include computations over the target program execution history. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog to the user as well. An assertion is a boolean expression that may contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or violated. A debugging rule has the form:

> assertion **SAY** (expression sequence)
>
> **ONFAIL SAY** (expression sequence)

The presence of metavariables in the assertion makes it possible to use FORMAN as a debugger query language. The computation of an assertion is interrupted when it becomes clear that the final value will be False, and the current values of metavariables can be used to generate readable and informative messages.

The following examples have been executed on our prototype FORMAN/PASCAL assertion checker [2], [3]. The

PASCAL program reads a sequence of integers from file XX.TXT.

```pascal
program e1;

    var X: integer;

    XX: file of text;

begin

    X:= 7;

    (* initial value is assigned here *)

    reset (XX, 'XX.TXT');

    while X<>0 do

        read(XX, X)

end.
```

The contents of the file XX.TXT are as follows:

```
11 5 3 7 8 9 3 13 2 3 45 8 754 45567 0
```

Example of a Query 1. In order to obtain the history of variable X the following computation over event trace can be performed. The rule condition is TRUE, and is shown as a side effect the whole history of variable X.

TRUE

SAY ('The history of variable X is:'

    [D: destination IS X FROM execute_program APPLY VALUE(D)])

The [...] construct above defines a loop over the whole program execution trace (execute_program event). All events matching the pattern destination IS X are selected from the trace and the function VALUE is applied to them. The resulting sequence consists of values assigned to the X variable during the program execution.

When executed on our prototype the following output is produced:

```
Assertion #1 checked successfully...
```

```
The history of variable X is: 7 11 5 3 7 8 9 3 13 2 45 8 754 45567 0
```

Example of an Assertion 2. Let's write and check the assertion : *"The value of* variable X does not exceed 17."

```
FOREACH *S: ex_stmt CONTAINS (D: destination IS X) FROM execute_program
```

```
VALUE(D) < 17

  ONFAIL

     SAY('Value ' VALUE(D) 'is assigned to the variable X in stmt ')

     SAY(S)

     SAY('This is record #' CARD[ ex_read_stmt FROM PREV_PATH(S)] + 1 'in the
file XX.TXT')
```

We check the assertion for all events where the value of X may be altered. These are events of the type destina-tion which can appear within ex_assignment_stmt or ex_read_stmt events. In order to make error messages about assertion violations more informative we include the embracing event of the type ex_stmt. Metavariables S and D refer to those events of interest. When the assertion is violated for the first time, the assertion evaluation terminates and current values of metavariables can be used for message output. The value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text

Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of step counter.

Since we expect the assertion might be violated when executing a Read statement, it makes sense to report the record number of the input file xx.txt where the assertion is violated. The program state does not contain any variables which values could provide this information. But we can perform auxiliary calculations independently from the target program using FORMAN aggregate operations. In this particular case the number of events of the type ex_read_stmt preceding the interruption moment is counted. This number plus 1 (since the violation occurs when the read statement is executed) yields the number of an input record on which the variable X was first assigned the value exceeding 17.

```
Assertion # 2 violation!

 Value 45 is assigned to the variable X in stmt

ex_stmt :> Read( XX , X )     Time= 73 .. 78

This is record # 11 in the file XX.TXT
```

Example of a Query 3. Profile measurement. In order to obtain the actual number of statements executed, the following query can be performed:

```
TRUE

SAY('The total number of statements executed is:'

    CARD[ ALL ex_stmt FROM execute_program ])
```

The ALL option in the aggregate operation indicates that all nested events of the type ex_stmt should be taken

into account.

```
    Assertion #3 checked successfully...
  The total number of statements executed is: 18
```

Example of a *generic assertion* which must be true for any program in the target language.

"Each variable has to be assigned value before it is used in an expression evaluation."

FOREACH * S: ex_stmt FROM execute_program

      FOREACH * E: eval_expression CONTAINS (V: variable) FROM S

           EXISTS D: destination FROM PREV_PATH(E) SOURCE_TEXT(D) = SOURCE_TEXT(V)

ONFAIL

      SAY('In event' S)

      SAY('in expression evaluation')

      SAY(E)

      SAY('uninitialized variable' SOURCE_TEXT(V) 'is used')

For the following PASCAL program our prototype detects the presence of the bug described above.

```
program e2;

var X,Y: integer;

begin    Y:= 3;

    if Y < 2 then begin

        X:= 7; Y:= Y + X

    else Y:= X - Y (*** here the error appears: X has no value! ***)
end.
```

```
Assertion #4 violation!

In event ex_stmt :> If ( Y < 2 ) then X := 7 ; Y := ( Y + X ) ;

                                     else Y := ( X - Y ) ; Time= 10 .. 35

in expression evaluation

eval_expression :> ( X - Y ) Time= 20 .. 29

uninitialised variable X is used
```

Debugging rules can be considered as a way of formalizing reasoning about the target program execution -- humans often use similar patterns for reasoning when debugging programs. For example, if the index expression of an array element is out of the range, the debugger can try a rule for eval-index events that invokes another rule about wrong value of the event eval-expression, which in turn will cause investigation of histories of all variables included in the expression.

Yet another application of generic assertions and debugging rules may be for describing run-time constraints (sequences of procedure calls, actual parameter dependences, etc.) for nontrivial subroutine packages, e.g. for the MOTIF package for GUI design. A library containing assertions and debugging rules relevant to such a package may be useful for writing C programs calling subroutines from the package.

4 Conclusions

In brief, our approach can be explained as "computations over a target program event trace." We expect the advantages of our approach to be the following:

- The notion of **an event grammar** provides a general basis for program behavior models. In contrast with previous approaches, the **event** is not a point in the trace but an interval with a beginning and an end.

- Event grammar provides a coordinate system to refer to any interesting event in the execution history. Program variable values are attributes of an event's beginning and end. Event attributes provide complete **access to each target program's execution state**. Assertions about particular execution states as well as assertions about sets of different execution states may be checked.

- The PRECEDES relation yields a **partial order** on the set of events, which is a natural model for parallel program behavior.

- The IN relation yields a **hierarchy of events**, so the assertions can be defined at an appropriate level of granularity.

- A language for **computations over event traces** provides a **uniform framework** for assertion checking, profiles, debugging queries, and performance measurements.

- The access to the complete target program execution history and the ability to formalize **generic assertions** can be used in order to define **debugging rules and strategies.**

- The fact that assertions and other computations over target program event trace can be **separated from the text of the target program** allows accumulation of formalized knowledge about particular programs and about the whole target language in separate files. This makes it easy to control the amount of assertions to be checked.

According to [7] and [12] approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e. bugs which could be detected by appropriate assertion checking similar to the demonstrated above.

References

[1] M. Auguston, "FORMAN -- A Program Formal Annotation Language", *Proceedings of the 5:th Israel Conference on Computer Systems and Software Engineering*, Gerclia, May 1991, IEEE Computer Society Press, 149-154.

[2] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.

[3] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", *in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995.

[4] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering*, vol.6, No 4, 1996, pp. 609-640.

[5] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE'97, Madrid, Spain, June 1997, pp. 257-262

[6] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *The Journal of Systems and Software* 3, 1983, pp. 255-264.

[7] B. Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[8] F. Brooks, The Mythical Man-Month, 2nd edition, Addison-Wesley, 1995.

[9] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.

[10] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software* 25, 1994, pp. 223-239.

[11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, No. 7, July 1978, pp. 558-565.

[12] S. L. Pfleeger, Software Engineering, Theory and Practice, Prentice Hall, 1998.

Building Program Behavior Models

Mikhail Auguston
Computer Science Department, New Mexico State University,
Las Cruces, NM 88003, USA
phone: (505) 646-5286, fax: (505) 646-1002
mikau@cs.nmsu.edu
http://www.cs.nmsu.edu/~mikau

Abstract. This paper suggests an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. Assertions can capture both the dynamic properties of a particular target program and can formalize the general knowledge of typical bugs and debugging strategies. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation. Event grammars may be designed for sequential as well as for parallel programs. The approach suggested can be adjusted to a variety of programming languages.

Keywords. Program behavior models, events, event grammars, software testing and debugging automation

1   Introduction

Dynamic program analysis is one of the least understood activities in software development. A major problem is still the inability to express the mismatch between the expected and the observed behavior of the program on the level of abstraction maintained by the user [11]. In other words, a flexible and expressive specification formalism is needed to describe properties of the software system's implementation. Program testing and debugging is still a human activity performed largely without any adequate tools and consuming more than 50% of the total program development time and effort [10]. Debugging concurrent programs is even more difficult because of parallel activities, non-determinism and time-dependent behavior.

One way to improve the situation is to partially automate the debugging process. Precise *model of program behavior* becomes the first step towards debugging automation. It appears that traditional methods of programming language semantics definition don't address this aspect.

In building such a model several considerations were taken in account. The first assumption we make is that the model

is discrete, i.e. comprises a finite number of well-separated elements. This assumption is typical for Computer Science methods used for static and dynamic analysis of programs. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) don't require the total ordering of actions, so *partial event ordering* is the most adequate method for this purpose [18].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

Finally, the program execution can be modeled as a set of events (*event trace*) with two basic relations: partial ordering and inclusion. The event trace actually is a model of program's behavior temporal aspect. In order to specify meaningful program behavior properties we have to enrich events with some attributes. An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of event), etc.

The next problem to be addressed after the program behavior model is set up is the formalism specifying properties of the program behavior. This could be done in many different ways, e.g. by adopting some kind of logic calculi (predicate logic, temporal logic). Such a direction leads to tools for program static verification, or in more pragmatic incarnations to an approach called model checking [13]. As indicated in [1] "Dynamic analysis is limited to checking observed behaviors, and so in principle provides weaker assurances, but this is balanced by checking a wider range of properties and typically

by better performance ... ."

Since our goal is debugging automation, i.e. a kind of program dynamic analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we came up with the concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN [2], [4], [16] based on functional paradigm and the use of event patterns and aggregate operations over events.

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data and control flows in the target program. Assertions can also be used as conditions in rules which describe debugging actions. For example, an error message is a typical action for a debugger or consistency checker. Thus, it is also possible to specify debugging strategies.

The notions of event and event type are powerful abstractions which make it possible to write assertions independent of any target program. Such generic assertions can be collected in standard libraries which represent the general knowledge about typical bugs and debugging strategies and could be designed and distributed as special software tools.

FORMAN is a general language to describe computations over program event trace that can be considered as an example of *a special programming paradigm*. Possible application areas include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [6]. A study of FORMAN application for parallel programming is presented in [5]

2   Events

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a message is sent or received, a statement is executed, or some expression is evaluated. A particular action may be per-

formed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

In order to give some support for our notion of event let us consider a well-known idea such as a counter. Usually the history of a variable X when used as a counter looks like:

X := 0; ...

Loop ...

    X := X + 1; ...

endloop; ...

In order to check whether the actual behavior of the counter X matches the pattern described by the program fragment above we have to consider the following events. Let Initialize_X denote the event of assigning 0 to the variable X, Augment_X denote the event of incrementing X, and Assign_X denote an event of assigning any value to the variable X. One could check whether X behaves as a counter when a program segment S is executed in the following way. First, the sequence A of all events of the type Assign_X from the event trace of program segment S has to be extracted preserving the ordering between events. Second, A has to be matched with the pattern:

```
Initialize_X     (Augment_X)  *
```

where '*' denotes repetition zero or more times. If the actual sequence of events does not match this pattern we can report an error. Therefore, assertion checking can be represented as a kind of computation over target program event trace.

Another informal example involves parallel events. Let us suppose that Assign_Y denotes an event of assigning a value to the shared variable Y through any of several parallel processes. Then, detecting a set of events of the type Assign_Y

AI2000.doc

that happen "at the same time" (i.e. are not under the precedence relation) may be evidence of a possible data-race condition in the program execution.

The program state (current values of variables) can be considered at the beginning or at the end of an appropriate event. This provides the opportunity to write assertions about program variable values at different points in the program execution history.

Program profiling usually is based on counting the number of events of some type, e.g. the number of statement executions or procedure calls. Performance measurements may be based on attaching the duration attribute to such events and summarizing durations of selected events.

3 Event Trace and the Language for Computations Over Event Traces

FORMAN is a high-level specification language for expressing intended behavior or known types of error conditions when debugging or testing programs. It is intended to be used in conjunction with a high-level programming language which is called the *target language*.

The model of target program behavior is formally defined through a set of general axioms about two basic relations, which may or may not hold between two arbitrary events: they may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more than one of these relations can be established.

There are several general axioms that should be satisfied by any events a, b. c in the event trace of any target program.

1) Mutual exclusion of relations.

```
a PRECEDES b  => not (a IN b) and not (b IN a)
a IN b  =>  not(a PRECEDES b) and not (b PRECEDES a)
```

2) Noncommutativity.

```
a PRECEDES b  =>  not( b PRECEDES a)
a IN b  =>  not( b IN a)
```

3) Transitivity.

```
(a PRECEDES b ) and ( b PRECEDES c ) => ( a PRECEDES c)
(a IN b ) and ( b IN c ) => ( a IN c)
```

Irreflexivity for PRECEDES and IN follows from 2). Note that PRECEDES and IN are irreflexive partial orderings.

4) Distributivity

```
(a IN b) and (b PRECEDES c) => (a PRECEDES c)

(a PRECEDES b) and (c IN b) => (a PRECEDES c)

(FOR ALL a IN b (FOR ALL c IN d (a PRECEDES c) )) => (b PRECEDES d)
```

In order to define the behavior model for some target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object and the grammar describes how the event is split into other event sequences or sets. For example, the event execute-assignment-statement contains a sequence of events evaluate-right-hand-part and execute-destination. The evaluate-right-hand-part, in turn, consists of an unique event evaluate-expression. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different type in the program execution history, it is not intended to be used for parsing actual event trace.

The rule A :: (B C) establishes that if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C, also exist, such that the relations b IN a, c IN a, b PRECEDES c hold.

For example, the event grammar describing the semantics of a PASCAL subset may contain the following rules. The

names, such as `execute-program,` and `ex-stmt` denote event types.

```
execute-program  :: ( ex-stmt * )
```

This means that each event of the type `execute-program` contains an ordered (w.r.t. relation PRECEDES) sequence of zero or more events of the type `ex-stmt`.

```
ex-stmt :: (  label? ( ex-assignment | ex-read-stmt | ex-write-stmt |

    ex-reset-stmt | ex-rewrite-stmt | ex-close-stmt | ex-cond-stmt |

    ex-loop-stmt | call-procedure) )
```

The event of the type `ex-stmt` contains one of the events `ex-assignment,` `ex-read-stmt,` and so on. This inner event determines the particular type of statement executed and may be preceded by an optional event of the type `label` (traversing a label attached to the statement).

```
ex-assignment :: (ex-righthand-part destination)
```

The order of event occurrences reflects the semantics of the target language. When performing assignment statement first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, for instance, source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, duration of the event, previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of target language to write assertions on target program variables [3] [4].

Events can be described by patterns which capture the structure of event and context conditions. Program paths can be

described by regular path expressions over events.

The main extension for the parallel case [5] consists of the introduction of a new kind of composite event -- "snapshot," which can be considered as an abstraction for the notion "a set of events that may happen at the same time." The "snapshot" event is a set of events each pair of which is not under the relation PRECEDES and makes it possible to describe and to detect at run-time such typical parallel processing faults as data races and deadlock states.

All this makes it possible to formalize assertions of the following types:

- "all variables in the program must be initialized before using in some expression."

- "file must be opened, then the read statement is performed zero or more times and after that the close statement is executed,"

- "at least one variable changes its value during one loop L iteration,"

- "after the execution of a subprogram P the value of variable X remains unchanged,"

- "there is an attempt to assign values to the same variable in two parallel processes" (data race condition).

- "deadlock for parallel processes P1 and P2 is detected."

In addition to debugging and testing, FORMAN can also be used to specify profiles and performance measurements.

4   Examples of Debugging Rules and Queries

In general, a *debugging rule* performs some actions that may include computations over the target program execution history. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog to the user as well. An assertion is a boolean expression that may contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or

AI2000.doc

violated. A debugging rule has the form:

```
assertion    SAY (expression sequence)

        ONFAIL SAY (expression sequence)
```

The presence of metavariables in the assertion makes it possible to use FORMAN as a debugger query language. The computation of an assertion is interrupted when it becomes clear that the final value will be False, and the current values of metavariables can be used to generate readable and informative messages.

The following examples have been executed on our prototype FORMAN/PASCAL assertion checker [3], [4]. The PAS-CAL program reads a sequence of integers from file XX.TXT.

```
program e1;
    var X: integer;
    XX: file of text;
begin
    X:= 7;
    (* initial value is assigned here *)
    reset (XX, 'XX.TXT');
    while X<>0 do
        read(XX, X)
end.
```

The contents of the file XX.TXT are as follows:

```
11 5 3 7 8 9 3 13 2 3 45 8 754 45567 0
```

Example of a Query 1. In order to obtain the history of variable X the following computation over event trace can be

performed. The rule condition is TRUE, and is shown as a side effect the whole history of variable X.

    TRUE

    SAY ('The history of variable X is:'

        [D: destination IS X FROM execute_program APPLY VALUE(D)]   )

The [...] construct above defines a loop over the whole program execution trace (execute_program event). All events matching the pattern destination IS X are selected from the trace and the function VALUE is applied to them. The resulting sequence consists of values assigned to the X variable during the program execution.

When executed on our prototype the following output is produced:

Assertion #1 checked successfully...

The history of variable X is: 7 11 5 3 7 8 9 3 13 2 45 8 754 45567 0

Example of an Assertion 2. Let's write and check the assertion : *"The value of* variable X does not exceed 17."

FOREACH *S: ex_stmt CONTAINS (D: destination IS X) FROM execute_program VALUE(D) < 17

ONFAIL

    SAY('Value ' VALUE(D) 'is assigned to the variable X in stmt ')

    SAY(S)

    SAY('This is record #' CARD[ex_read_stmt FROM PREV_PATH(S)] + 1 'in the file XX.TXT')

We check the assertion for all events where the value of X may be altered. These are events of the type destination which can appear within ex_assignment_stmt or ex_read_stmt events. In order to make error messages about assertion violations more informative we include the embracing event of the type ex_stmt. Metavariables S and D refer

AI2000.doc

to those events of interest. When the assertion is violated for the first time, the assertion evaluation terminates and current values of metavariables can be used for message output. The value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text

Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of step counter.

Since we expect the assertion might be violated when executing a Read statement, it makes sense to report the record number of the input file xx.txt where the assertion is violated. The program state does not contain any variables which values could provide this information. But we can perform auxiliary calculations independently from the target program using FORMAN aggregate operations. In this particular case the number of events of the type ex_read_stmt preceding the interruption moment is counted. This number plus 1 (since the violation occurs when the read statement is executed) yields the number of an input record on which the variable X was first assigned the value exceeding 17.

```
Assertion # 2 violation!

Value 45 is assigned to the variable X in stmt

ex_stmt :> Read( XX , X )      Time= 73 .. 78

This is record # 11 in the file XX.TXT
```

Example of a Query 3. Profile measurement. In order to obtain the actual number of statements executed, the following

AI2000.doc

query can be performed:

TRUE

```
SAY('The total number of statements executed is:'

    CARD[ ALL ex_stmt FROM execute_program ])
```

The ALL option in the aggregate operation indicates that all nested events of the type ex_stmt should be taken into

account.

```
    Assertion #3 checked successfully...

  The total number of statements executed is: 18
```

Example of a *generic assertion* which must be true for any program in the target language.

"Each variable has to be assigned value before it is used in an expression evaluation."

FOREACH * S: ex_stmt FROM execute_program

    FOREACH * E: eval_expression CONTAINS (V: variable) FROM S

        EXISTS D: destination FROM PREV_PATH(E) SOURCE_TEXT(D) = SOURCE_TEXT(V)

ONFAIL

    SAY('In event' S)

    SAY('in expression evaluation')

    SAY(E)

    SAY('uninitialized variable' SOURCE_TEXT(V) 'is used')

For the following PASCAL program our prototype detects the presence of the bug described above.

```
program e2;

var X,Y: integer;

begin     Y:= 3;

    if Y < 2 then begin
```

AI2000.doc

```
            X:= 7; Y:= Y + X


    else Y:= X - Y (*** here the error appears: X has no value! ***)

  end.
```

Assertion #4 violation!

In event ex_stmt :> If (Y < 2) then X := 7 ; Y := (Y + X) ;

                                                    else Y := (X - Y) ; Time= 10 .. 35

in expression evaluation

eval_expression :> (X - Y) Time= 20 .. 29

uninitialised variable X is used

Debugging rules can be considered as a way of formalizing reasoning about the target program execution -- humans often use similar patterns for reasoning when debugging programs. For example, if the index expression of an array element is out of the range, the debugger can try a rule for eval-index events that invokes another rule about wrong value of the event eval-expression, which in turn will cause investigation of histories of all variables included in the expression.

Yet another application of generic assertions and debugging rules may be for describing run-time constraints (sequences of procedure calls, actual parameter dependences, etc.) for nontrivial subroutine packages, e.g. for the MOTIF package for GUI design. A library containing assertions and debugging rules relevant to such a package may be useful for writing C programs calling subroutines from the package.

5   Related Work

What follows is a very brief survey of basic ideas known in Debugging Automation to provide the background for the

approach advocated in this paper.

5.1   Event Notion

The Event Based Behavioral Abstraction (EBBA) method suggested in [8] characterizes the behavior of the whole program in terms of both primitive and composite events. Context conditions involving event attribute values can be used to distinguish events. EBBA defines two higher level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering serves to eliminate from consideration events which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

An event-based debugger for the C programming language called Dalek [23] provides a means for description of user-defined events which typically are points within a program execution trace. A target program has to be instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events.

FORMAN has a more comprehensive modelling approach than EBBA or Dalek, based on the event grammar. A language for expressing computations over execution histories is provided, which is missing in EBBA and Dalek. The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events. FORMAN supports design of universal assertions and debugging rules that could be used for debugging of arbitrary target programs. This generality is missing in EBBA and Dalek approaches. The event in FORMAN is a time interval, in contrast with the event notion in previous approaches where events are considered pointwise time moments.

The COCA debugger [14] for the C language uses the GDB debugger for tracing and PROLOG for debugging queries execution. It provides a certain event grammar for C traces and event patterns based on attributes for event search. The query language is designed around special primitives built into the PROLOG query evaluator. We assume that FORMAN is more suitable for trace computations as it has been designed for this specific purpose.

5.2   Path Expressions

Data and control flow descriptions of the target program are essential for testing and debugging purposes. It is useful to

give such a description in an explicit and precise form. The path expression technique introduced for specifying parallel programs in [12] is one such formalism. Trace specifications also are used in [22] for software specification. This technique has been used in several projects as a background for high-level debugging tools, (e.g. in [11]), where path rules are suggested as kinds of debugger commands. FORMAN provides flexible language means for trace specification including event patterns and regular expressions over them.

5.3 Assertion Languages

Assertion (or annotation) languages provide yet another approach to debugging automation. The approaches currently in use are mostly based on boolean expressions attached to selected points of the target program, like the assert macro in C. The ANNA [20] annotation language for the Ada target language supports assertions on variable and type declarations. In the TSL [19], [25] annotation language for Ada the notion of event is introduced in order to describe the behavior of Tasks. Patterns can be written which involve parameter values of Task entry calls. Assertions are written in Ada itself, using a number of special pre-defined predicates. Assertion-checking is dynamic at run-time, and does not need post-mortem analysis. The RAPIDE project [21] provides a reach event-based assertion language for software architecture description.

In [7] events are introduced to describe process communication, termination, and connection and detachment of process to channels. A language of Behavior Expressions (BE) is provided to write assertions about sequences of process interactions. BE is able to describe allowed sequences of events as well as some predicates defined on the values of the variables of processes. Event types are process communication and interactions such as send, receive. terminate, connect, detach. Evaluation of assertions are done at run-time. No composite events are provided.

Another recent experimental debugging tool is based on trace analysis with respect to assertions in temporal interval logic. This work is presented in [17] where four types of events are introduced: assignment to variables, reaching a label, interprocess communication and process instantiation or termination. Composite events cannot be defined. Different varieties of temporal logic languages are used for program static analysis called Model Checking [13].

In [26] a practical approach to programming with assertions for the C language is advocated, and it is demonstrated that even local assertions associated with particular points within the program may be extremely useful for program debug-

ging.

The FORMAN language for computations over traces provides flexible means for writing both local and global assertions, including those about temporal relations between events.

5.4   Algorithmic Debugging

The original algorithmic program debugging method was introduced in [28] for the Prolog language. In [27] and [15] this paradigm is applied to a subset of PASCAL.

The debugger executes the program and builds a trace execution tree at the procedure level while saving some useful trace information such as procedure names and input/output parameter values. The algorithmic debugger traverses the execution tree and interacts with the user by asking about the intended behavior of each procedure. The user has the possibility to answer "yes" or "no" about the intended behavior of the procedure. The search finally ends and a bug is localized within a procedure p when one of the following holds: procedure p contains no procedure calls, or all procedure calls performed from the body of procedure p fulfill the user's expectations.

Algorithmic debugging can be considered as an example of debugging strategy, based on some assertion language (in this case assertions about results of a procedure call.) The notion of computation over execution trace introduced in FORMAN may be a convenient basis for describing such debugging strategies.

6   Conclusions

In brief, our approach can be explained as "computations over a target program event trace." We expect the advantages of our approach to be the following:

- The notion of **an event grammar** provides a general basis for program behavior models. In contrast with previous approaches, the **event** is not a point in the trace but an interval with a beginning and an end.

- Event grammar provides a coordinate system to refer to any interesting event in the execution history. Program variable values are attributes of an event's beginning and end. Event attributes provide complete **access to each target program's execution state**. Assertions about particular execution states as well as assertions about sets of different

execution states may be checked.

- The PRECEDES relation yields a **partial order** on the set of events, which is a natural model for parallel program behavior.

- The IN relation yields a **hierarchy of events**, so the assertions can be defined at an appropriate level of granularity.

- A language for **computations over event traces** provides a **uniform framework** for assertion checking, profiles, debugging queries, and performance measurements.

- The access to the complete target program execution history and the ability to formalize **generic assertions** can be used in order to define **debugging rules and strategies.**

- The fact that assertions and other computations over target program event trace can be **separated from the text of the target program** allows accumulation of formalized knowledge about particular programs and about the whole target language in separate files. This makes it easy to control the amount of assertions to be checked.

According to [9] and [24] approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e. bugs which could be detected by appropriate assertion checking similar to the demonstrated above.

It appears that the approach initially designed for program behavior modeling may be used in other dynamic system behavior models as well. The methodology is based on identifying event types representing essential actions performed within the system, and defining the basic relations PRECEDES and IN for those events (event grammar), and appropriate event attributes. Then the FORMAN-like language for computations over event traces may be developed to specify behavior properties, to perform queries and other kinds of dynamic analysis.

References

[1]   F. Anger, R. Rodriguez, M. Young, "Combining Static and Dynamic Analysis of Concurrent Programs", *Proceedings of International IEEE Conference on Software Maintenance,* Victoria. BC, Canada, Sept. 1994, pp.89-98.

AI2000.doc

[2] M. Auguston, "FORMAN -- A Program Formal Annotation Language", *Proceedings of the 5:th Israel Conference on Computer Systems and Software Engineering*, Gerclia, May 1991, IEEE Computer Society Press, 149-154.

[3] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.

[4] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", *in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995.

[5] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering*, vol.6, No 4, 1996, pp. 609-640.

[6] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE'97, Madrid, Spain, June 1997, pp. 257-262

[7] F. Baiardi, N. De Francesco, G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering*, vol. SE-12, No. 4, April 1986, pp. 547-553.

[8] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *The Journal of Systems and Software* 3, 1983, pp. 255-264.

[9] B. Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[10] F. Brooks, The Mythical Man-Month, 2nd edition, Addison-Wesley, 1995.

[11] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.

[12] R. H. Campbell, A. N. Habermann, "The specification of process synchronization by path expressions", *Lecture Notes in Computer Science*, vol. 16, 1974, pp. 89-102.

AI2000.doc

[13] E.Clarke et al., "Verification tools for Finite State Concurrent Systems", LNCS vol.803, 1994, pp.124-175.

[14] M.Ducasse, "COCA: An automated debugger for C", in Proceedings of the 1999 International Conference on Software Engineering, ICSE 99, Los Angeles, 1999, pp.504-513.

[15] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing", *ACM LOPLAS -- Letters of Programming Languages and Systems.* Vol. 1, No. 4, December 1992.

[16] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software* 25, 1994, pp. 223-239.

[17] G. Goldszmidt, S. Katz, S. Yemini, "Interactive Blackbox Debugging for Concurrent Languages", *SIGPLAN Notices* vol. 24, No. 1, 1989, pp. 271-282.

[18] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, No. 7, July 1978, pp. 558-565.

[19] D. C. Luckham, D. Bryan, W. Mann, S. Meldal, D. P. Helmbold, "An Introduction to Task Sequencing Language, TSL version 1.5" (Preliminary version), Stanford University, February 1, 1990, pp. 1-68.

[20] D. C. Luckham, S. Sankar, S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications", *IEEE Software*, January 1991, pp.74-84.

[21] D. Luckham, J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering. Vol.21, No. 9, 1995, pp. 717-734.*

[22] J. McLean, "A Formal Method for the Abstract Specification of Software", *Journal of the Association of Computing Machinery*, vol.31, No. 3, July 1984, pp. 600-627.

[23] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", *Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.*

[24] S. L. Pfleeger, Software Engineering, Theory and Practice, Prentice hall, 1998.

AI2000.doc

[25] D. Rosenblum, "Specifying Concurrent Systems with TSL", *IEEE Software*, May 1991, pp.52-61.

[26] D. Rosenblum, "A Practical Approach to Programming With Assertions", *IEEE Transactions on Software Engineering, Vol. 21, No 1, January 1995, pp. 19-31.*

[27] N. Shahmehri, "Generalized Algorithmic Debugging", Ph.D. Thesis No. 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1991.

[28] E. Shapiro, "Algorithmic Program Debugging", MIT Press, May 1982.

Static Analysis for Program Generation Templates[1]

Valdis Berzins
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

This paper presents an approach to achieving reliable cost-effective software via automatic program generation patterns. The main idea is to certify the patterns once, to establish a reliability property for all of the programs that could possibly be generated from the patterns. We focus here on properties that can be checked via computable static analysis. Examples of methods to assure syntactic correctness and exception closure of the generated code are presented. Exception closure means that a software module cannot raise any exceptions other than those declared in its interface.

1. Introduction

Our goal is to provide cost effective means for creating reliable software. We are addressing the issue by improving the technology for automatic software generation, with particular attention to reliability issues.

We take a domain specific view of this process: a domain is a family of related problems addressing a common set of issues. A domain analysis identifies the problem and issues, formulates a model of these, and determines a corresponding set of solution methods. Users of the proposed computer-aided software generation system describe their particular problem using a domain specific problem modeling language that provides concrete representations of problems in the domain. The system then automatically determines which solution methods are applicable, customizes them to the specific problem instance described using the modeling language, and then automatically generates a program that will solve the specified problem.

We seek to provide tool support for the above process that can be applied to many different problem domains, and that can generate code in any programming language. Therefore we seek uniform and effective methods for generating software generators of the type described above, given definitions of the problem modeling language, the target programming language, and the roles for synthesizing solution programs. A simple architecture for this process is shown in Figure 1.

The specific goals of this paper are: (1) to provide a simple example of a language for expressing software patterns that are specific enough to be used as synthesis rules and (2) to provide examples of static rules in this language. We address the problems of certifying that all programs which can be generated from a given set of rules: (1) are syntactically correct and (2) will not raise any exceptions other than those explicitly specified in an interface description.

This is a step towards a coordinated system of static and dynamic checks, to be performed on program synthesis rules. Our hypothesis is that the most cost effective way to improve software quality is to systematically improve and certify the rules used to generate a domain-specific software generator. This approach directly addresses the issue of correctly implementing given software requirements. It also indirectly addresses the issue of getting the right requirements, because it should eventually enable rapid prototyping of product quality systems by problem domain experts, who need not be software experts. If the requirements are found to be inappropriate, the domain experts will simply update the problem models and regenerate a new version of the solution software.

We will refer to the software generation patterns as templates. Our rationale for the claim of cost effectiveness is that the benefits of quality improvements to the templates can be extended to all past and future applications of the generators - by regenerating the generator using the improved templates and then regenerating the past applications. The regeneration process can be completely automated, thereby reducing labor costs, eliminating a source of random human errors, and speeding up the process of repairing a known fault throughout a large family of software systems.

The relation to the theme of this workshop is that fast moving scenarios can be addressed by automatically generating new variants of the software that reflect changing issues in the problem domain. Our approach should reduce the explicit quality assurance efforts needed each time the software is changed. By amortizing the quality assurance effort applied to the template over many applications of the same templates, we can reduce quality assurance costs. The benefits increase with the number of systems generated from the same templates.
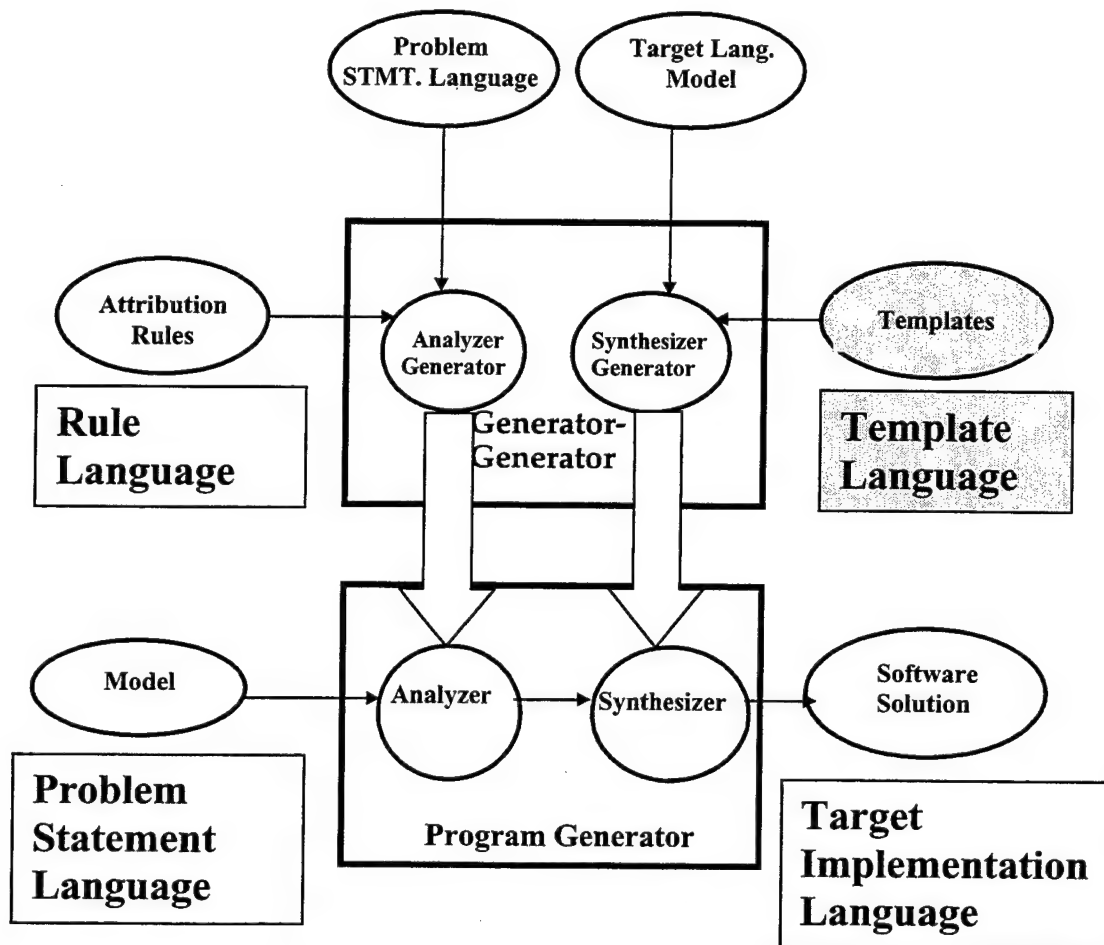
Figure 1. Model-Based Software Generator Architecture

This paper focuses on static checks that can be completely automated. Our research is also addressing testing and debugging of program synthesis rules and proofs of rule properties that require human assistance with deeper reasoning. These efforts are outside the scope of the current paper, which is organized as follows:

- Section 2 formalizes software generation patterns and defines a uniform construction to obtain a template language for any target programming language.
- Section 3 describes methods for statically certifying syntactic correctness generated code, and gives an example.
- Section 4 does the same for analysis of exceptions.
- Section 5 contains comparisons to previous work
- Section 6 presents conclusions.

2.   Template Languages

The purpose of a template language is to define software synthesis patterns for a given target language. We create such languages based on a functional object model of code generation templates. We take a functional (i.e. side-effect-free) approach because this simplifies the algebraic basis of the approach and supports effective static analysis methods such as those presented in Section 3 and 4.

We view template languages as extensions of the corresponding target programming languages. Because many different programming languages are created, we will need many different template languages. However, all of these can be defined at once by providing uniform construction such as that shown in Figure 2.

This is a very simple construction, but it is very expressive. In addition to providing substitution of actual values for generic parameters, as in the generic units of Ada and the templates of C++, our construction includes conditionals that are evaluated at code generation time, and the ability to invoke other templates. Recursion is included.

Template_language = {template, formal_def, template_expression}

DEF_TEMPLATE(id[template], type, seq[formal_def], template_expression):
      template   -- where type $^\circ$ ∈ `target_language`

DEF_FORMAL(template_parameter, type): formal_def
      -- declares the type of a formal parameter

template_parameter < {id[any], template_expression}
IF(template_expression, template_expression, template_expression):
      template_expression
APPLY(id[template], seq[template_expression]): template_expression

template_expression < `target_language`

Figure 2. Template Abstract Syntax

The construction depends heavily on the use of inheritance in object-oriented modeling of programming languages. The situation is illustrated in Figure 3.
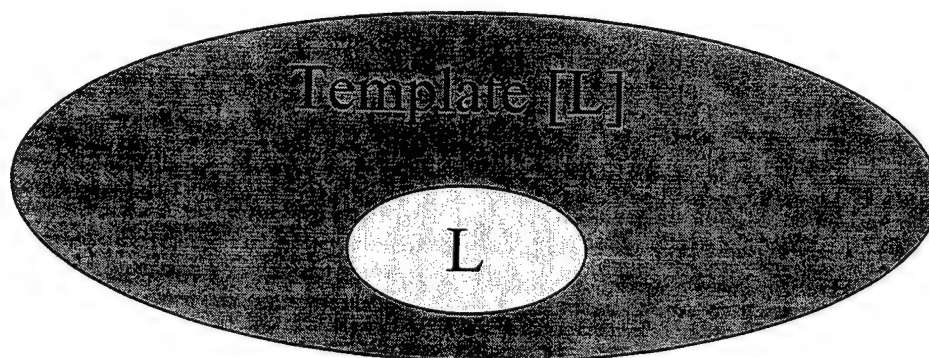
Figure 3. Generic Template Language

In object-oriented modeling, class-wide types[2] are viewed as open and extensible. Specifically, each time we add a subclass with a new constructor, we add more instances to the class-wide type, thus extending its value set.

We model the abstract syntax of a language using a type for each kind of semantic entity. In a properly constructed abstract syntax, there should be one such type for each non-terminal symbol. Each constructor of these types corresponds to a production of the grammar. Subclass relationships, denoted by "≤", specify that every instance of the subclass is also an instance of the parent class. Multiple inheritance is allowed. For example, in line 6 of Figure 2 says that every template parameter is a kind of identifier, and also is a kind of template expression. This kind of subclass relationship is used to incorporate reusable types in a library of programming language building blocks, such as identifiers, and to specialize reusable concepts to the application, such as template expression. If T is a type and S is a set of types, T<S means T is a subclass of each element of S. This represents multiple inheritance.

---

[2] This is Ada 95 terminology. The instances of a class wide type include its direct instances and those of all its subclasses, transitively.

Subclassing is also used to interface between a target programming language and its extensions. In Figure 2, "target-language" denotes the set of types comprising the abstract syntax of the target language. Figure 4 shows a very simple example of a target language that illustrates how this works.

```
target_language = (stmt, exp)

assign(var, exp): stmt
if(exp, stmt, stmt): stmt

integer < exp    -- integer literals
var < {id[any], exp} -- program variables
apply(id[function], seq[exp]): exp -- operations

subtype rule:  x < y ==> id[x] < id[y] where x, y ∈ type
```

Figure 4. Example: Micro Target Language

The example in Figure 5 defines a code generation pattern that embodies Newton's method for polynomial evaluation, which is optimal in terms of number of evaluation steps needed. This is a very simple example of a code generation pattern that is nevertheless realistic, because it embodies a solution method. The example also illustrates the use of all the constructs in the template language. We use infix syntax for the exp constructors * and + to improve legibility (e.g. x*y is short for the term apply(*, x, y)).

An additional benefit of considering the abstract syntax to be an algebra rather than a tree is that we can used well-studied transformation rules. In particular we can associate equational axioms with the programming language types that define normal forms. Figure 5 illustrates the use of such axioms as rewrite rules that simplify the code produced by the generator in a follow-on normalization process. This is one way to incorporate optimizations into the program generation process, which is useful for unconditional transformations.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
    -- c contains coefficients of a polynomial, lowest degree first
    IF not (is_empty (c) )   -- use operations of boolean and seq
    THEN v * (evaluate_polynomial (v, rest(c))) + first (c)
    ELSE 0
END TEMPLATE

Template application evaluate-polynomial(x, [1, 2, 3]) generates
    x * (x * (x * 0 + 3) + 2) + 1

Normalization with integer rules i * 0 = 0, i + 0 = i reduces to
    x * (x * 3 + 2) + 1
```

Figure 5. Example: Generation Pattern

Code generation using the template language is a very much like evaluation in a functional programming language with call-by-value semantics. Analysis of templates can take advantage of equational reasoning, substitution, and structural induction. The limitation to primitive recursion facilitates the latter. The recursion in the example is structural because **rest** is a partial inverse for the sequence constructor **add** (i.e. rest(add(x, s)) = s).

3.   Syntactic Correctness of Generated Code

We treat the abstract syntax structures of the target language as the values of the abstract data types representing the programming language. We require these types to provide a pretty printing operation that outputs such objects as text strings according to the concrete syntax of the target language, with a readable format. Establishing correctness of these pretty printing operations is straightforward, and in fact their implementations can be generated from an appropriately annotated grammar for the concrete syntax.

Given trusted pretty printing operations for the object model of the target language, syntactic correctness of the output reduces to the type-correctness of the ground terms generated by the evaluation

of the templates. This can be checked using a simple type system for the template language and conventional type checking methods. Note that we are referring to the types associated with the signatures of the constructors in the object model of the target programming language, rather than the types within the target programming language, which may not even be a typed language. The process is illustrated Figure 6. The computed type annotations are shown in *italics*. The type annotations associated with the implicit induction step, where the type signature of the template itself is used, is highlighted in ***bold italics***. The indentations of the type annotations reflect the structure of the derivation.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
  IF not (is_empty (c : seq[integer]  ) : boolean ) : boolean
  THEN  + ( * ( v                                          : var,
           evaluate_polynomial
             (v                    : var,
              rest(c: seq[integer] ) : seq[integer])  : exp
                                        )  : exp
           first (c: seq[integer] )            : integer
                                        )  : exp
        —term form of v* evaluate_polynomial (v, rest(c)) + first (c)
  ELSE 0                                                 : integer
  END TEMPLATE
```

Types conform because integer < exp and var < exp

Relevant signatures: +(exp, exp) :exp, *(exp, exp) :exp,
  first(seq[T]): T, rest(seq[T]): seq[T],
  is_empty(seq[T]): boolean, not(boolean): boolean

Figure 6. Example: Syntactic Correctness of Generated Code

Note that induction has been carried out implicitly, as a routine step of the type checking calculation. This is sufficient to establish partial type correctness of the templates, which implies syntactic correctness of all code that could be generated by the template, it does not automatically guarantee total correctness, because we still have the possibility that evaluation of the template might fail to terminate.

Total correctness is established by the type check if we check that all recursions are primitive. The example satisfies this condition because rest is a partial inverse of the compound sequence constructor; rest(add(x,s)) = s. This means that the induction is in fact structural, and hence that evaluate_polynomial is total. Thus the template will produce syntactically correct code for all input values that conform to the type signature of evaluate_polynomial.

We note that given declarations of the target language constructors that define the abstract syntax and the corresponding partial inverse operations, it is straightforward to automatically check that all recursive calls are primitive with respect to any given parameter position. This implies that structural induction can be applied uniformly and completely automatically in this context. Furthermore, our experience suggests that structural recursions are sufficient to define the code generation templates needed in practice, and that template designers can live within the restriction to structural recursions without undue hardships.

4. Exception Closures for Generated Code

One common source of software failure is unhandled exceptions. This section explains a method for certifying that all programs generated from a given template cannot generate any unhandled exceptions when placed in a context that handles a specified set of exceptions.

Our approach is to refine the type system to record the set of exceptions that might be raised by the evaluation of any expression of the target language. A similar structure can be used to analyze the set of exceptions that might be raised by execution of a statement of the target language.

The refinement replaces the single target language type exp with a parameterized family of types exp[set[exception]]. The intended interpretation of this type structure is that evaluation of an expression of type exp[S] might raise an exception e only if e∈ S. Since we do not require all exceptions in S to be producible, this family of types has a rich subclass structure defined by the following relation:

$$S1 \subseteq S2 \Rightarrow exp[S1] \leq exp[S2]$$

The type signatures of an operation are specified explicitly for argument expression type that cannot raise any exceptions, and are extended to all other types by the following rule, which describes the essential pattern for propagating exceptions:

$$F(exp[\varnothing]) : exp[S1] \Rightarrow f(exp[S2]): exp[S1 \cup S2]$$

The rule for operations with multiple arguments is similar. Similar rules apply to language constructs representing exception handlers. Exception handlers follow rules of the form

$$(TRY\ exp[S1]\ CATCH\ e\ USE\ exp[S2]): exp[(S1-\{e\}) \cup S2].$$

Figure 7 shows the exception analysis for our running example. The parts added to the version in Figure 6 are underlined.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp [{ovfl}]
IF not (is_empty (c:  seq[integer]boolean boolean
   THEN +(*(v:  var
           evaluate_polynomial(v: var,
               rest(c:  seq[integer]seq[integer] ):  exp [{ovfl}]
           first (c:  seq[integer]integer exp{ovfl}]
   -- term form of  v * evaluate_polynomial (v, rest(c)) + first (c)
   ELSE 0:  integer
END TEMPLATE
```

Types conform because integer < exp [∅] ≤ exp [{ovfl}] and
var < exp [∅] ≤ exp [{ovfl}]

Relevant signatures: +(exp, exp): exp [{ovfl}] , *(exp, exp): exp [{ovfl}] ,
first(seq[T]): T, rest(seq[T]): seq[T], is_empty(seq[T]): boolean, , not(boolean): boolean

Figure 7. Exception Closure of Generated Code

Note that we require the author of the template to specify in the type declaration of a template the set of exceptions the generated expression is allowed to raise. This acts as an induction hypothesis in our exception analysis, which is used when analyzing the recursive call of evaluate-polynomial. It also provides useful information for the user of the generated code.

The analysis shown in the figure establishes a partial exception closure: it guarantees that all expressions generated by the template can at most raise only the exception **ovfl** representing integer overflow.

To establish a total exception closure, we have to address clean termination of the template expansion at program generation time. The primitive recursion check explained in the previous section guarantees there will be no infinite recursions, so that termination is guaranteed. However, for clean termination, we must also check that evaluation of the template will not raise any exceptions at program generation time.

Note that the analysis in Figure 7 addresses run-time exceptions. When viewed as constructors of the abstract syntax, + and * are total operations. Overflow exceptions can occur only when those expressions are evaluated, not when they are constructed.

61

The sequence operators **first** and **rest** are different: they are partial query methods of the abstract syntax, not total constructors. If applied to an empty sequence, they raise a sequence underflow exception. However, this can occur only at program generation time, not at run time.

To certify clean termination of template at program generation time requires a type refinement to record sets of possible exceptions and an additional kind of type refinement to record domains of partial methods such as **first** and **rest**. We can introduce a subtype nseq[T, S] < seq[T, S] consisting of the nonempty sequences, and refine the signatures of the partial sequence operations first and rest as follows.

$$first(nseq[T, \varnothing]): T[\varnothing] , rest(nseq[T, \varnothing]): seq[T, \varnothing]$$
$$first(seq[T, \varnothing]): T[seq\_underflow], rest(seq[T, \varnothing]): seq[T, \{seq\_underflow\}]$$

Type analysis requires a bit of inference in this case, because we have to use the guard of the template language conditional IF together with the rule

$$s : seq[T, S] \text{ and not is-empty } (s) \Rightarrow s: nseq[T, S]$$

This inference is easy because the guard matches the subtype restriction predicate for nseq[T].

This match did not occur by accident - the purpose of the guard is precisely to ensure that the operations first and rest are used only within their domain of definition. In the interests of being able to produce certifiably robust code, we claim that it would not be unduly burdensome to require that template designers associate domain predicates with all partial operations, and use those domain predicates explicitly in guards whenever they are needed to ensure the partial operators are used within their proper domains of definition. For example, **first** could be associated with a domain predicate

$$first\text{-}ok (seq[T]) : boolean \quad where$$
$$first\text{-}ok (s) = not (is\text{-}empty (s)).$$

This would enable a fast and shallow analysis of guard conditions to certify absence of exceptions in cases like this. Some such restriction is necessary for practical engineering support because the problem of checking whether an unconstrained guard condition implies the domain predicates of arbitrary guarded partial operations is undecidable.

An alternative is an exception analysis that includes exceptions in the closure even in cases where the guard condition ensures they will never arise. We suggest that it is more practical to handle a common subset of efficiently recognizable forms, and to ask designers to work within the constraints of those recognizable forms. We believe this would be less burdensome than the alternative of manually analyzing the cases where a type check insensitive to guard conditions would nominate exceptions that cannot in fact occur, and that it would lead to a more robust software by making it practical to do complete analysis of exception closures. For example, we could require the example of Figure 7 to be written in a stylized form that looks like the following:

IF first-ok (c) and rest-ok (c)
THEN ... **first** (c) ... **rest** (c) ...

A similar type check would have to be applied to the implementations of **first** and **rest** to ensure that they would in fact terminate cleanly whenever the domain predicates are true.

5.   Comparisons to Previous Work

One of our contributions has been to formalize and abstract the idea of a program generation pattern, to make it independent of the details of the target programming language and the process of instantiating the patterns. The purpose of this was to create context in which systematic analysis of program generation patterns becomes possible and in some cases becomes decidable.

Program generation patterns have been evolving for a long time. Macros are an early form of the idea. However, macros are notoriously difficult to analyze, partially because they traditionally operate on uninterpreted text. This makes the connection between macro definitions and the behavior they ultimately denote complicated and potentially very indirect. The macros in LISP are an improvement because they are based on abstract syntax trees rather than characters. However, in this context a second source of complexity becomes apparent: a macro can expand to produce another macro, and the number

62

The sequence operators **first** and **rest** are different: they are partial query methods of the abstract syntax, not total constructors. If applied to an empty sequence, they raise a sequence underflow exception. However, this can occur only at program generation time, not at run time.

To certify clean termination of template at program generation time requires a type refinement to record sets of possible exceptions and an additional kind of type refinement to record domains of partial methods such as **first** and **rest**. We can introduce a subtype nseq[T, S] < seq[T, S] consisting of the nonempty sequences, and refine the signatures of the partial sequence operations first and rest as follows.

$$first(nseq[T, \varnothing]): T[\varnothing] , rest(nseq[T, \varnothing]): seq[T, \varnothing]$$
$$first(seq[T, \varnothing]): T[seq\_underflow], rest(seq[T, \varnothing]): seq[T, \{seq\_underflow\}]$$

Type analysis requires a bit of inference in this case, because we have to use the guard of the template language conditional IF together with the rule

$$s : seq[T, S] \text{ and not is-empty } (s) \Rightarrow s: nseq[T, S]$$

This inference is easy because the guard matches the subtype restriction predicate for nseq[T].

This match did not occur by accident - the purpose of the guard is precisely to ensure that the operations first and rest are used only within their domain of definition. In the interests of being able to produce certifiably robust code, we claim that it would not be unduly burdensome to require that template designers associate domain predicates with all partial operations, and use those domain predicates explicitly in guards whenever they are needed to ensure the partial operators are used within their proper domains of definition. For example, **first** could be associated with a domain predicate

$$first\text{-}ok \ (seq[T]) : boolean \quad where$$
$$first\text{-}ok \ (s) = not \ (is\text{-}empty \ (s)).$$

This would enable a fast and shallow analysis of guard conditions to certify absence of exceptions in cases like this. Some such restriction is necessary for practical engineering support because the problem of checking whether an unconstrained guard condition implies the domain predicates of arbitrary guarded partial operations is undecidable.

An alternative is an exception analysis that includes exceptions in the closure even in cases where the guard condition ensures they will never arise. We suggest that it is more practical to handle a common subset of efficiently recognizable forms, and to ask designers to work within the constraints of those recognizable forms. We believe this would be less burdensome than the alternative of manually analyzing the cases where a type check insensitive to guard conditions would nominate exceptions that cannot in fact occur, and that it would lead to a more robust software by making it practical to do complete analysis of exception closures. For example, we could require the example of Figure 7 to be written in a stylized form that looks like the following:

$$IF \ first\text{-}ok \ (c) \ and \ rest\text{-}ok \ (c)$$
$$THEN \ ... \ \textbf{first} \ (c) \ ... \ \textbf{rest} \ (c) \ ...$$

A similar type check would have to be applied to the implementations of **first** and **rest** to ensure that they would in fact terminate cleanly whenever the domain predicates are true.

5.   Comparisons to Previous Work

One of our contributions has been to formalize and abstract the idea of a program generation pattern, to make it independent of the details of the target programming language and the process of instantiating the patterns. The purpose of this was to create context in which systematic analysis of program generation patterns becomes possible and in some cases becomes decidable.

Program generation patterns have been evolving for a long time. Macros are an early form of the idea. However, macros are notoriously difficult to analyze, partially because they traditionally operate on uninterpreted text. This makes the connection between macro definitions and the behavior they ultimately denote complicated and potentially very indirect. The macros in LISP are an improvement because they are based on abstract syntax trees rather than characters. However, in this context a second source of complexity becomes apparent: a macro can expand to produce another macro, and the number

63

of expansion steps before the generated source code actually appears is potentially unbounded. This makes the system very difficult to analyze. At the other extreme are the generic units of Ada. These are strongly typed, clearly connected to the abstract syntax of the language, and the results of instantiating them are easy to analyze. However, they do not allow conditional decisions at instantiation time, and are restricted in the sense that the abstract syntax trees of all possible instantiations have exactly the same shape, up to substitution for the formal parameters of the pattern. A language-independent version of the idea can be found in [5], although this appears to be largely text-based.

Another aspect of our approach is to model languages as algebras rather than as abstract syntax trees. A hint of this idea appears in [4], although it is not exploited there for enabling analysis to any significant degree. The work of the CIP group [1] develops this idea further and takes advantage of the reasoning structures that come with the algebraic modeling approach, such as term rewriting and generation induction principles. This suggests extension to a full object-oriented view, which includes inheritance. The Refine system is the earliest context we know of where grammars are treated as object models with potential inheritance structures, although the documentation does not give any hint about the significance of this capability. In this paper we demonstrate the usefulness of algebraic models of syntax with inheritance, for defining language extension transformations that can be applied to all possible target languages.

Another theme is lightweight inference [2]. We have demonstrated that some useful types of static analysis for program generation patterns can be performed via computable and indeed reasonably efficient methods. The processes described here can be implemented using technologies typically used in compilers, such as object attribution rules, they terminate for all possible inputs, and do so in polynomial time. We believe this approach will scale up to large applications, and are currently working out the details to support a tight analysis of the efficiency of the process.

This paper has explored static analysis of meta-programs to check syntactic correctness and exception closure of the generated code. Another kind of static analysis in this family, type checking of meta-programs to ensure the type correctness of the generated code, is considered by another paper in this proceedings [3].

6. Conclusions

We believe that formal models of program generation templates can support a variety of quality improvement processes that can help achieve cost-effective software reliability. This paper has presented a simple example of such a formal model and two such quality improvement processes, certification of syntactic correctness and freedom from unexpected exceptions for all programs that can be generated from a given program generation pattern. We expect the greatest advantages of this approach to be realized when it is applied to realize flexible and reliable systems in a product line approach. This approach should be augmented with systematic methods for domain analysis that culminates in the development of a domain-specific library of solutions embodied in a domain-specific software architecture that is populated with components produced by model-based software generators. When the technology matures, it should become possible for problem domain experts to specify their problem instances in terms of familiar problem domain models, and to have reliable software solutions to their problems automatically generated, without direct involvement of computer experts.

The economic advantage of this approach comes from the ability to automatically reap the benefits of each quality improvement for all past and future instantiations of the template (if past applications are regenerated). We believe that it will be profitable to explore methods for lifting many known program analysis techniques from the level of individual programs to the level of program generation patterns. This should be explored for a variety of issues that range from certifying absence of references to uninitialized variables, absence of deadlock, and many others, perhaps ultimately to template-based proof of post conditions and program termination for generated programs.

To make this vision practical, many engineering issues must be addressed, including presentation issues, methods for lightweight inference [2] and support for transforming and enhancing complex sets of analysis rules. Other issues include systematic methods for dynamic analysis, testing, and debugging of program generation rules. It is not reasonable to expect progress to occur in an instantaneous quantum leap to perfection. A realistic process is a gradual one, where simple sets of program generation rules are deployed, and gradually tuned, improved, certified, and extended. A key issue is enabling rule enhancement and exception closure extension without invalidating all previous effort on analysis and certification of the previous versions.

The difference between the program generation approach proposed here and current compiler generation tools is the associated static analysis capabilities for the program generation rules. It is possible that in the future, ultra-reliable compilers will be built using techniques derived from those introduced in this paper.

REFERENCES

1. F. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Paukner and P. Pepper, *The Munich Project CIP*. Vol. 2: The Program Transformation System CIP-S, Springer, Berlin, 1987.

2. V. Berzins, Light Weight Inference for Automation Efficiency, Proceedings of the *1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Monterey California, 1999.

3. N. Bjorner, Type Checking Meta Programs, Proceedings of the Workshop on *Modeling Software System Structures in a Fastly Moving Scenario*, Santa Margherita, Italy, 2000.

4. T. Reps, *Generating Language-Based Environments*, Doctoral Dissertation, August 1982.

5. D. Volpano, R. Kieburtz, Software Templates, CS/E 85-011, Department of Computer Science and Engineering, Oregon Graduate Center, 1985.

HIGH LEVEL NET MODELS: A TOOL FOR PERMUTATION MAPPING AND FAULT DETECTION IN MULTISTAGE INTERCONNECTION NETWORK

Nabendu Chaki
Department of Computer Science & Engineering
University of Calcutta
92 A P C Roy Road
Calcutta 700 091, INDIA
email : nabendu@ieee.org

Swapan Bhattacharya
Department of Computer Science & Engineering
Naval Post-Graduate School
Montery
CA, USA
email : swapan@cs.nps.navy.mil

Abstract: This paper aims at structurising detection of different types of Stuck-at faults for a wide range of Multistage Interconnection Networks (MINs). The results reported so far in this respect are mainly based on direct combinatorial analysis of the concerned networks with very little consideration towards the modelling aspects. The graphical representation coupled with well-defined semantics allowing formal analysis has already established Petri Net as an effective tool for modelling dynamic systems. However, the existing variants of high level nets had certain limitations in modelling the dynamic behaviour of mapping a permutation through MIN and further analysis of the same. This has inspired the authors to propose a couple of new high level net model, called MP-net and S-net in their earlier works. The S-net model uses tokens to hold and propagate information apart from controlling firing of events. It uses two different types of places and transitions each as has been defined subsequently. In this paper, we have concentrated in detection of fault in MINs using this S-net model.

Keywords
Petri Net, MIN, Stuck-at-fault, S-net, Data Place, Control Place

I. Introduction

Generalized Stochastic Petri Net (GSPN) is a performance analysis tool [11] based on the graphical system representation typical of Petri nets, in which some transitions are timed, while others are immediate. Distributed, parallel and real time systems may be modelled using this GSPN. However, for any large system comprising of large number of components the time distributions and relations between components are often quite complex [07, 08]. This largeness and complexity is reflected in the corresponding GSPN models.

The capability of incorporating time as a parameter in net based models have been taken care of with the introduction of Time Petri Nets [13] and Timed Petri Nets [12, 14, 15]. The Timed Petri nets are derived from Petri nets by associating a finite firing duration with each transition. The classical firing rule of Petri nets is thus modified to account for the time taken to fire a transition

and also to express that a transition must fire as soon as it is enabled. Time Petri Nets (TPN) are more general in the sense that a Timed Petri net can be modelled by using Time Petri net, but the reverse is not true. For both of these models, firing of a transition is a non-atomic operation. The firing is said to be in progress in between a start firing event and an end-firing event.

In the context of MINs, binary values are used to represent information pertaining to data as well as control. A study of different variants of high level nets, as discussed above, indicates that for modelling different processing elements of distributed computation, some additional flexibility is to be incorporated in the basic modelling tool to take care of variations in structures and functionality of these hardware elements.

In the Modified Petri Net (MP-net) model [03], as defined by us earlier, two different types of Places and Transitions are used [03]. The MP-net model for a NxN network consisting of $O(log_2N)$ stages would involve $O(Nlog_2N)$ number of subnets, one each for every 2x2 cross-bar switch that constitute the MIN. The total number of Data and Control places as well as the number of Controlled transitions will therefore be $O(Nlog_2N)$. This would lead to an unmanageable and complex situation for the description of a large system. Thus it has been felt that the proposed MP-net model requires further compactness. The Stochastic behavior of MP-net is coupled with the properties of Colored Petri net [10] to propose a new powerful high level net called S-net. It has been achieved by equipping each token with an attached data value called the Token color. In S-net, there has been a significant improvement in total number of places as well transitions comparing Mp-net. Both redundant path MINs like Benes and non-redundant path MINs like Omega or Baseline have been modelled using S-net [01][03].

Essentially, a variant of Coloured and Stochastic nets, the S-net has been established as an ideal tool for modelling any element that has to handle two different types of signals in repetitive, modular units. It has been already used to model different types of MINs, e.g. Omega, Benes, Baseline networks, etc [03]. It has also been found that

66

using S-net, a MIN of NxN size can be modelled with $3N/2$ number of places only as against $O(N\log_2 N)$ number of switching elements for the corresponding MIN. As far as studying the MINs, the results reported so far [02], [04], [07] are mainly based on direct combinatorial analysis of the concerned networks with very little consideration towards the modelling aspects. In the present paper, we have concentrated in detection of fault in MINs using this S-net model. The definition of S-net and some of the relevant terminologies that are essential to understand the actual problem of permutation mapping and fault detection using the model are presented in the following section.

II. Definition and Terminology

2.1 S-net Model

An S-net model uses two different types of places and transitions each that enables it to handle data and control signals as two separate entities. S-net is represented by a seven-tuple $\{D, C, P, T_c, T_i, I, O\}$, where,

$D = \{d_i : d_i \text{ is a Data place}\}$;
A data place holds exactly one token in it at some instance. The token value is a positive integer that indicates the information held by the element being modelled. A token stored in Data place is not used to decide the flow of Control. A Data place is always safe.
$C = \{c_i : c_i \text{ is a Control place}\}$;
A control place holds token to enable corresponding Controlled transition for firing. A token value is represented by an ordered pair $<x,y>$ where x represents token color and y is the control value, typically 0 and 1 for two logical states. The number of tokens in a Control place must not exceed the number of different colors used in the model and there can be only one token for a particular color in a single Control place. A Control place is k-bounded, where the maximum number of colors in the model is k.
$P = \{p : p \text{ is a color}\}$;
$T_c := \{t : t \text{ is a Controlled transition}\}$;
A Controlled transition can have one and only one Control place at its input and the transition is enabled and fired in presence of some token of the same color as that for the current stage, having some pre-defined control value in the corresponding input Control place.
$T_i := \{t : t \text{ is an Immediate transition}\}$;
An Immediate transition is enabled and fired irrespective of the presence of token in its input place. In fact, none of the input places for an Immediate transition is a Control place. An Immediate transition is fired in between a start time and an end time in a stochastic manner.
$I = \{T_c, T_i\} \rightarrow D^\infty$ is the input function, a mapping from transitions to bags of Input places.
$O = \{T_c, T_i\} \rightarrow D^\infty$ is the Output function, a mapping from transitions to bags of Output Data places.

Different sets of places and transitions as specified in the definition of S-net are disjoint. Unlike MP-net, in the proposed S-net model, the same Data places are to hold

different data values for different colors as indicated by some member of the Color set P. Similarly, the token value to be stored in a Control place depends on the color. The firing rule for the two types of transitions are very similar to that for the MP-net except that incase of S-net, the color of the token is considered. Whenever a Controlled transition is fired in color p, tokens in its input Data places are transferred to corresponding output Data places following the directed arcs. The token in the Control place of color value p is removed after the Controlled transitions are fired. On the other hand, an Immediate transition connecting an input Data place D_k to an output Data place D_m for color p transfers the token of D_k to D_m on its firing.

2.2 Properties of S-net

The S-net $(D, C, P, T_c, T_i, I, O)$ has been defined to structerise the performance analysis of MPP systems and some of its subsystems with the help of modelling through it. Before this, in the present section some of the basic properties of S-net has been discussed.

2.2.1 Marking : The presence of token values in places, at an instance, is called marking of the S-net model. There will be two separate sets of markings $\bullet_D(d_0, d_1, .., d_D)$ for D Data places and $\bullet_C(c_0, c_1, .., c_C)$ for C number of Control places such that d_k for $k \in [1..D]$ is some positive integer a if the corresponding Data place holds a token of value a. On the other hand marking of a Control place c_k for $k \in [1..C]$ is a set of ordered pairs $<a,b>$, where a is the token color and b is the control value.

In case of a 2x2 cross-bar switch, the control value is either 0 or 1, indicating the through and crossed states of the switch respectively. The number of elements in the set of ordered pairs c_k must not exceed the number of different colors used in the model and there can be only one token of a particular color in a single Control place. As a Controlled transition is fired in a color p, the token of the same color in its input Control place is perished. Thus, after the last set of Controlled transitions of in an S-net model is fired, all the Control places are found empty.

2.2.2 Initial State Definition : The initial state of a S-net is defined as a marking of Data and Control places. Initially all the Data places being used in a model holds one token each. In case a NxN multistage interconnection network is being modelled, the input permutation is stored as the initial state of the set of N Data places. A Control place, on the other hand, is initialised with k different tokens (k≤m), for maximum m number of colors being used in the model. A controlled transition is enabled at color p, if the corresponding input Control place is initialized with a token of color p with appropriate control value.

2.2.3 Boundedness : A place in a net is Safe if the number of tokens in that place never exceeds one The Data places in S-net are therefore Safe, by definition. Actually,

Safeness is a special case of the more general Boundness property. A place is said to be k-safe or k-bounded if the number of tokens in that place never exceed an integer value of k. A place that is 1-safe is simply called Safe place. Therefore the Control places used in a S-net, are p-bounded, where the maximum number of colors in the model is p. As the all places in a S-net are bounded, the S-net itself is bounded.

2.2.4   Reachability : The initial marking of the Data places are changed as different Controlled and Immediate transitions are fired in different colors in an S-net model. A state $\bullet_D$ is said to be reachable if a particular sequence of firing of Controlled and Immediate transitions exists following which the initial marking of Data places is modified to $\bullet_D$. A reachability set may be defined as the set of all markings reachable from the initial marking. The reachability analysis on an S-net model may be performed without direct consideration of the changes in marking of the Control places.

III.   Permutation mapping and Control matrix

A permutation P consists of several individual transmissions in between two extreme input/output lines at the opposite ends of the MIN. A Control place may hold a token signifying the crossed state of corresponding 2x2 switch. For example, let's consider that the Control place C1 holds a token in pass 1. This would result in shifting content of Data place D1 into D2 at the end of pass 1. Before the second pass begins, the time independent transitions cascading the basic blocks, are fired. This would take the original content of D1 onto D3. Thus after the execution of second pass, the content of first input line of the MIN will finally be mapped onto the third or fourth output line, depending upon the content of C2 at pass 2.
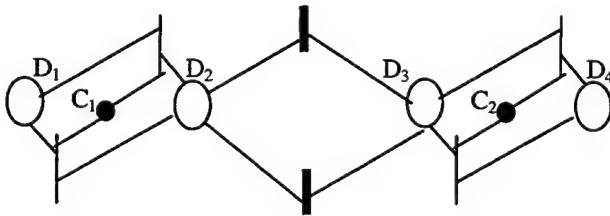
(Figure 1 : S-net model for a 4x4 Omega Network)

Thus, it may be inferred that presence of a token in C1 for pass 1 and absence of a token in C2 for pass 2 enforces a transmission in between input line 1 and output line 3 of the Omega network under consideration.

The effect of presence or absence of a token in C1 for pass 2 and in C2 for pass 1 can be neglected. A matrix representation $\begin{pmatrix} 1 & x \\ x & 0 \end{pmatrix}$ may be proposed to depict the state of the model for the particular link. Here **1** represents presence of a token in the corresponding Control place, **0**

represents absence of token in the same and **x** is a don't care symbol. A matrix like the one presented above may be termed as Control matrix. For any NxN MIN, the dimension of the Control matrix would be (kxm) where k represents number of passes and m is the number of Control places in the model, which in any case would be n/2.

An entry $\{C_{ij} : i \bullet [1...k] , j \bullet [1...m]\}$ in the Control matrix reflects the presence or absence of a token in Control place C_j for pass i. Considering a few links, the corresponding Control matrices for those are presented below :

$$\text{link } 1 \to 3 : \begin{pmatrix} 1 & x \\ x & 0 \end{pmatrix} \qquad \text{link } 3 \to 2 : \begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix}$$

$$\text{link } 1 \to 4 : \begin{pmatrix} 1 & x \\ x & 1 \end{pmatrix} \qquad \text{link } 3 \to 4 : \begin{pmatrix} x & 1 \\ x & 0 \end{pmatrix}$$

$$\text{link } 2 \to 1 : \begin{pmatrix} 1 & x \\ 0 & x \end{pmatrix} \qquad \text{link } 4 \to 2 : \begin{pmatrix} x & 1 \\ 0 & x \end{pmatrix}$$

$$\text{link } 2 \to 2 : \begin{pmatrix} 1 & x \\ 1 & x \end{pmatrix} \qquad \text{link } 4 \to 4 : \begin{pmatrix} x & 0 \\ x & 0 \end{pmatrix}$$

Thus, mapping of every individual transmissions can be followed using the proposed model. This is in line with the fact that that there exists a path in between every pair of input-output lines for an Omega Interconnection network. But Omega, being a blocking MIN, in a conflict free situation, the Control matrix for the entire permutation, which essentially is a group of n individual links, may be derived with the help of the Control Matrixes for individual links.

Let's consider a permutation (2314) to be mapped using the proposed S-net model for Omega MIN. The transmission links involved are thus $(1 \to 3)$, $(2 \to 1)$, $(3 \to 2)$ and $(4 \to 4)$. The respective Control matrices for the links are to be considered simultaneously to identify conflicts, if any. A conflict here may be of the type that for two different transmissions, a particular C_{ij} position is found to contain 1 in the Control matrix for some link and 0 in the Control matrix for some other for any two values of i and j where i $\bullet$ [1...k] and j $\bullet$ [1...m]. The Control Matrix for the permutation (2314) through Omega Network will be $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$.

IV.   Detection of Fault using S-net

The present paper aims at efficient detection of different types of faults using the proposed S-net or its variation. Corresponding to every stage of a MIN, there will be an expected output pattern for a set input pattern. Thus an input pattern $(p_1 \ p_2 \ p_3 \ ... \ p_n)$ gets modified at different stages of a MIN as it is mapped through the same. The different stuck at faults or complete failure of one or more

constituent crossbar switch(s) can be detected by observing the actual output patterns off the stages of the MIN and comparing it to the corresponding expected output pattern. In the event that these two patterns are not identical, presence of fault(s), its type and position may be detected. But in case the expected and actual patterns are the same, certain types of stuck at fault(s) may still be there in individual switching elements. An algorithm has been proposed to detect faults and their positions in such cases.

However, without using a High level net like the S-net model, all the input and output links of each and every switch is to be checked to detect fault(s). Thus $O(Nlog_2N)$ links are to be checked all together to detect all possible faults in a NxN MIN. Further, to detect fault in redundant and non-redundant type of MINs different approaches are to be adopted. The following algorithm extracts the main advantage of having only $O(N)$ representative data places for all the $O(Nlog_2N)$ links of a MIN quite efficiently. Thus detection of various faults for a wide range of MIN becomes much easier by using S-net model of the MIN.

At a particular stage **k** of the MIN, the N/2 number of switches in the stage hold an input pattern (i_{k1} i_{k2} .. i_{kN}) at N input links. This is represented by the content of the N Data places for colour k before the Controlled Transitions are fired. Depending upon the content of the Control Matrix as mentioned above, the Controlled Transitions are fired and just before any of the Immediate Transitions for the pass are fired content of the Data places in the model reflect the expected output pattern (o_{k1} o_{k2} .. o_{kN}) off the stage **k**. One may find whether the actual output pattern (oa_{k1} oa_{k2} .. oa_{kN}) is same as (o_{k1} o_{k2} .. o_{kN}). A bitwise operation can detect some position q, where the actual content of the output link oa_{kq} and expected value o_{kq} are not the same. This indicates that the $\lceil q/2 \rceil^{th}$ switch at stage k of the MIN is faulty. A $O(Nlog_2N)$ algorithm has been presented below to describe the fault detection in MINs using the proposed S-net model.

Procedure DetectFault

Var
boolean flag
bit O[M][N], OA[M][N]
integer k,q;

Begin
  flag = .T.;
  For k=1 to M   /* M is $O(log_2N)$ represents the number
                  of stages in the MIN */
    For q=1 to N
      Derive O[k][q] from I[k][q] and Control matrix
      entry for pass k;
      If (O[k][q] $\oplus$ OA[k][q]) == 1 then
        Indicate fault in $\lceil q/2 \rceil^{th}$ switch of stage k;
        flag = .F.;
      Endif
      Fire Immediate transitions;
    Endfor
  Endfor
  If flag == .T. then
    Permutation may be mapped successfully;
  Endif
End

The algorithm presented above checks for faults that might block a particular permutation. This, however, does not ensure that the whole MIN is fault-free even if the *variable flag* is found to be .T. after the final iteration is over. For example, if a switch is having a stuck-at-T fault and at the same time if the corresponding Control matrix entry for permutation P is set to 0, the permutation can be successfully even in presence of the fault.

Thus for detection of multiple stuck at faults, in a non-redundant path network, the algorithm DetectFault is to be operated in two passes. In pass 1, all the control matrix entries are set to 0 whereby the Stuck-at-X, Stuck-at-U and Stuck-at-L faults are detected. In pass 2, all the control matrix entries are set to 1 and the Stuck-at-T, Stuck-at-U and Stuck-at-L faults are detected. Thus under fault-free condition, in pass 1, the Identity permutation should be realized, whereas in pass 2, the Complement permutation should be realized.

Similarly for Redundant path and Partially Redundant MINs as well, all types of faults can be identified with the help of S-net model. There are two basic advantages in using the S-net model. Firstly, the approach provides a Snapshot in the sense that for different colours the same Data places are representing the entire network. In stead of looking into the four input/output links of each of the $O(Nlog_2N)$ switches, the reliability of the MIN may be decided just by observing content of a fixed number of N Data places for any NxN network. This helps in designing a simple but efficient fault detection algorithm.
Apart from this, the introduction of Control matrix makes it more convenient to understand mapping of a permutation through different stages of a MIN. This is also quite helpful for any performance analysis of the network as instead of physically setting the crossbar switches with some control signal, the impact can be studied just by altering the corresponding Control matrix entry and then looking into the changes in the S-net model.

V.   Conclusion

The methodology for modelling MINs using the proposed S-net, has a wide range of applicability. These high level net models are designed to achieve optimum compactness so that analysis can be done more efficiently. Total number of Data and Control places in the proposed S-net models for any MIN is much less than that of 2x2 Cross-bar switches required to design the network itself. The number of switching elements required for a NxN MIN would be $O(Nlog_2N)$ whereas the corresponding S-net model would consist of only N number of Data places

and N/2 number of Control places. Moreover, for any NxN network, number of places is a constant linear function of N only. Modelling with S-net is thus quite effective for compact representation of Interconnection networks as well as for detecting different types of faults. In stead of looking into the input/output links of each of the $O(N\log_2 N)$ switches, the performance of a MIN may be studied and faults may as well be detected just by observing content of a fixed number of N Data places for any NxN network. Further, the introduction of Control matrix and the algorithm as discussed in section IV suggest that the present work may be extended to study and detect Stuck at faults in a wide range of MINs. It is, therefore, being proposed to consolidate this research work by taking care of the analysis of different Interconnection networks based on this model.

References

[01]   N Chaki, S Bhattacharya, "Permutation Mapping for MIN using High Level Net Models" Proc. of 1997 IEEE-ICPADS, Seoul, Korea, pp 548-555, December 10-13 1997.

[02]   S Bhattacharya, et.al., "Performance analysis of Omega Interconnection Network", 2nd IASTED International Conference on Computer Applications in Industry, Alexandria, Egypt, May 1992.

[03]   N Chaki, S Bhattacharya, "Modelling of Multistage Interconnection Networks using S-net", 16th IASTED Int'l Conference on Modelling and Simulation, Innsbruck, Austria, Feb. 1997

[04]   Xian Cheng and Oliver C Ibe, "Reliability of a class of Multistage Interconnection Networks", IEEE Transactions on Parallel and Distributed Systems Vol. 3, No 2, March 1992.

[05]   T. Y. Feng, "A survey of Interconnection Networks", IEEE Transactions on Computers Vol. 14, pp. 12-27, Dec. 1981.

[06]   C. L. Wu and T. Y. Feng, "On a class of multistage Interconnection Networks", IEEE Transactions on Computers, Vol. C-29, pp. 694-702, 1980.

[07]   J H Patel, "Performance of processor-memory interconnection networks for multiprocessors", IEEE Transactions on Computers, Vol C-30, pp 771-780, 1981.

[08]   T Murata, "Petri Nets : Properties, Analysis and Applications", IEEE Proceedings, Vol 77, pp 541-580, 1989.

[09]   K Jensen, "High level Petri Nets", DAIMI PB-151, September 1982.

[10]   Kurt Jensen, "Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use", Vol 1, Springer-Verlag, 1992.

[11]   Chiola G, Marson M A, Balbo G & Conte G, "Generalized Stochastic Petri Nets: A Definition of the net level and its implications", IEEE Transactions on Software Engg., Vol 19, No. 2, February 1993.

[12]   M A Holliday and M K Vernon, "A generalized timed Petri net model for Performance Analysis", IEEE Transactions on Software Engineering, Vol 12, pp 1297-1310, December 1987.

[13]   B Berthomieu and M Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets", IEEE Transactions on Software Engineering, Vol 17, pp 259-273, March 1991

[14]   C Ramachandani, "Analysis of asynchronous concurrent systems by Timed Petri nets", Massachusetts Institute Technol., Ph.D. Dissertation, 1974.

[15]   W Zuberek, "Timed Petri nets and preliminary performance evaluation", Proc. of 7th Annual Symposium on Computer Architecture, May 1980.

Automated Generation Of Wrappers For Interoperability

Ngom Cheng
cheng@cs.nps.navy.mil

Valdis Berzins
berzins@cs.nps.navy.mil

Luqi
luqi@cs.nps.navy.mil

Swapan Bhattacharya
swapan@cs.nps.navy.mil

Department of Computer Science
Naval Postgraduate School
833 Dyer Road
Monterey, CA. 93943 USA

Abstract

The major hurdle in developing distributed systems is the implementing the interoperability between the systems. Currently, most of the interoperability techniques require that the data or services to be tightly coupled to a particular server. Furthermore, as most programmers are trained in designing stand-alone application, developing distributed system proves to be time-consuming and difficult. This paper address the issues by creating an interface wrapper model that allows developers the features of treating distributed objects as local objects. A tool was developed to generate Java interface wrapper from a specification language called the Prototyping System Description Language.

1. Introduction

1.1 Background

Interoperability between software systems is the ability to exchange services from one system to another. In order to exchange services, data and commands are relayed from the service providers to the requesters. Current business and military systems are typically 2-tier or 3-tier systems involving clients and servers, each running on different machines in the same or different locations. Current approaches for n-tier systems have no standardization of protocol, data representation, invocation techniques etc. Other problems with interoperability are the implementation of distributed systems and the use of services from heterogeneous operating environments. These include issues concerning sharing of information amongst various operating systems, and the necessity for evolution of standards for using data of various types, sizes and byte ordering, in order to make them suitable for interoperation. These problems make interoperable applications difficult to construct and manage.

1.2 Current State-of-the-art solutions

Presently, the solutions attempting to address these interoperability problems range from low-level sockets and messaging techniques to more sophisticated middleware technology like object resource brokers (CORBA, DCOM). Middleware technology uses higher abstraction than messaging, and can simplify the construction of interoperable applications. It provides a bridge between the service provider and requester by providing standardized mechanisms that handle communication, data exchange and type marshalling. The implementation details of the middleware are generally not important to developers building the systems. Instead, developers are concerned with service interface details. This form of information hiding enhances system maintainability by encapsulating the communication mechanisms from the developers and providing a stable interface services for the developers. However, developers still need to perform significant work in incorporating the middleware's services into their systems. Furthermore, they must have a good knowledge of how to deploy the middleware services to fully exploit the features provided.

Current middleware approaches have another major limitation in the design - the data and services are tightly coupled to the servers. Any attempt to parallelize or distribute a computation across several machines therefore encounters complicated issues due to this tight control of the server process on the data.

1.3 Motivation

Distributed data structures provide an entirely different paradigm. Here, data is no longer coupled to any particular process. Methods and services that work on the data are also uncoupled from any particular process. Processes can now work on different pieces of data at the same time. So far, building distributed data structures together with their requisite interface has proved to be more daunting than other conventional interoperability middleware techniques. The arrival of JavaSpace has changed the scenario to some extent. It allows easy creation and access of distributed objects. However, issues concerning data getting lost in the network, duplicated data items, out-dated data, external exception handling and handshaking of communication between the data owner and data users are still open. The developers have to devise ways to solve those problems and standardize them between applications.

1.4 Proposal

The situation concerning interoperability would greatly improve if a developer working on some particular application were provided with the features capable of treating distributed objects as local objects within the application. The developers could then modify the distributed object as if it is local within the process. The changes may, however, still need to be reflected on other applications using that distributed object without creating any problems related to inconsistency. The current research aims at attaining this objective by creating a model of an interface wrapper that can be used for a variety of distributed objects. In addition, by automating the process of generating the interface wrapper directly from the interface specification of the requirement, developers' productivity is greatly improved.

The tools, named as Automated Interface Codes Generator (AICG), has been developed to generate the interface wrapper codes for interoperability, from a specification language called the Prototype System Description Language (PSDL) [LUQ88]. The tool uses the principle of distributed data structure and JavaSpace Technology to encapsulate transaction control, synchronization, and notification together with lifetime control to provide an environment that treats distributed objects as if there were local within the concerned applications.

2. Review of Existing Works

2.1 ORB Approaches

A basic idea for enhancing interoperability is to make the network transparent to the application developers. The existing approaches [1] include 1)Building blocks for interoperability, 2) Architectures for unified, systematic interoperability and 3) Packaging for encapsulating interoperability services. These approaches have been assessed using the Kiviat graphs by Berzins [1] with various weight factors. The Kiviat graphs give a good summary of the strong and weak points of various approaches. ORB and Jini are currently the more promising technologies for interoperability.

There are however, some concerns with the ORB models. Sullivan [13] provides a more in-depth analysis of the DCOM model, highlighting the architecture conflicts between Dynamic Interface Negotiation (how a process queries a COM services and interface) and Aggregation (component composition mechanism). The interface negotiation does not function properly within the aggregated boundaries. This problem arises because components share an interface. An interface is shared if the constructor or QueryInterface functions of several components can return a pointer to it. QueryInterface rules state that a holder of a shared interface should be able to obtain interfaces of all types appearing on both the inner and outer components. However, an aggregator can refuse to provide interfaces of some types appearing on an inner component by hiding the inner component. Thus, QueryInterface fails to work properly with respect to delegation to the inner interface.

Hence, for the ORB approaches, detailed understanding of the techniques is required to design a truly reliable interoperable system. Programmers however, are train mostly on standalone programming techniques. Adding specialized network programming models increases the learning as well as development time, with occasional slippage of target deadlines. Furthermore, bugs in the distributed programs are harder to detect and consequences of failure are more catastrophic. An abnormal program may cause other programs to go astray in a connected distributed environment [9], [12].

2.2 Prototyping

The demand for large, high quality systems has increased to the point where a quantum change in software technology is needed [9]. Rapid prototyping is one of the most promising solutions to this problem. Completely automated generation of prototype from a very high-level language is feasible and in-fact generation of skeleton programming structures is very common in the computer world. One major advantage of the automatic generation of codes is that it frees the developers from the implementation details by executing specification via reusable components [9].

In this perspective, an integrated software development environment, named Computer Aided Prototyping System (CAPS) has been developed at the Naval Postgraduate School, for rapid prototyping of hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, and military Command, Control, Communication and Intelligence (C3I) systems [11]. Rapid prototyping uses rapidly constructed prototypes to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process. The heart of CAPS is the Prototyping System Description Language (PSDL). It serves as an executable prototyping language at a specification or design level and has special features for real-time system design. Building on the success of computer aided rapid prototyping system (CAPS) [11], the AICG model also uses the PSDL for the specification and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view.

2.3 Transaction Handling

Building a networked application is entirely different from building a stand-alone system in the sense that many additional issues need to be taken care of for smooth functioning of a networked application. The networked systems are also susceptible to partial failures of computation, which can leave the system in an inconsistent state.

Proper transaction handling is essential to control and maintain concurrency and consistency within the system. Yang [16], examined the limitation of hard-wiring concurrency control (CC) into either the client or the server. He found that the scalability and flexibility of these configurations is greatly limited. Hence, he presented a middleware approach: an external transaction server, which carries out the concurrency control policies in the process of obtaining the data. Advantages of this approach are 1) transaction server can be easily tailored to apply the desired CC policies of specific client applications. 2) The approach does not require any changes to the servers or clients in order to support the standard transaction model. 3) Coordination among the clients that share data but have different CC policies is possible if all of the clients use the same transaction server.

The AICG model uses the same approach, by deploying an external transaction manager provided by SUN in the JINI model. All transactions used by the clients and servers are created and overseen by the manager.

3. The Basic Model

The AICG model is based on the concepts of encapsulating some of the features of the JavaSpace and Jini to provide a simplified ways of developing distributed applications. Section 3.1 examines the principles of JavaSpace and section 3.2 discusses some of the features of AICG model.
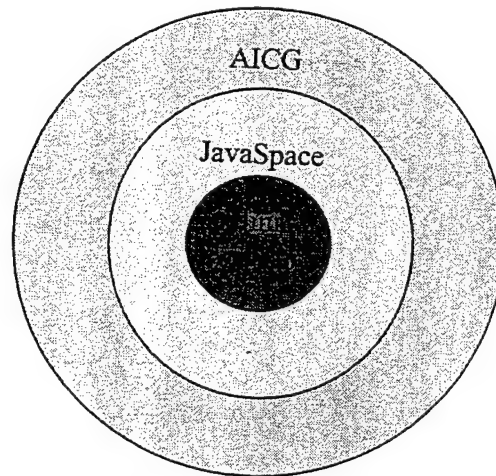
Figure 1, AICG Model

3.1 The JavaSpace Model

JavaSpace model is a high-level coordination tool for gluing processes together in a distributed environment. It departs from conventional distribution techniques using message passing between processes or invoking methods on remote objects. The technology provides a fundamentally different programming model that view an application as a collection of processes cooperating via the flow of freshly copied objects into and out of one or more spaces. This space-based model of distributed computing has its roots in the Linda coordination language [3] developed by Dr. David Gelernter at Yale University.
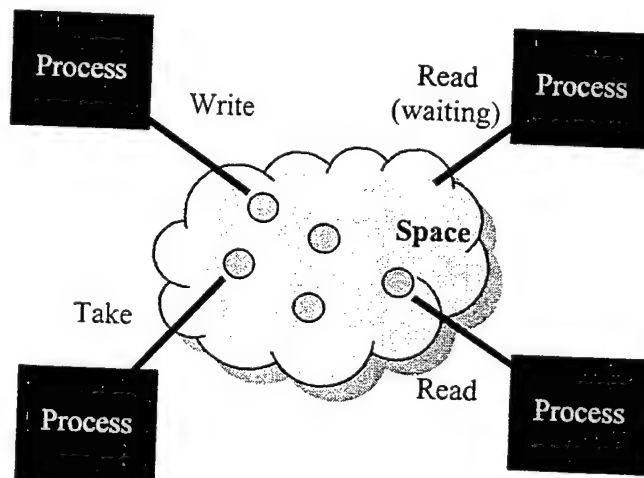
Figure 2, JavaSpace operations

A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism. As shown in figure 2, processes perform simple operations to write new objects into space, take objects from space, or read (make a copy of) objects in a space. When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. During the period of updating, other processes requesting for the object will wait until the process write the object back to the space.

Key Features of JavaSpace:
- Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it.
- Spaces are transactionally secure: The Space technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.
- Spaces allow exchange of executable content: While in the space, objects are just passive data, however, when we read or take an object from a space, a local copy of the object is created. Like any other local object, we can modify its public fields as well as invoke its methods.

3.2 The AICG Model

The AICG interoperability approach proposes a tool for building distributed applications. The tool is designed to generate interface wrappers for data structures or objects that need to be shared, and are particularly useful for applications that can model as flows of objects through one or more servers. Build on top of JavaSpace, the AICG model hides the space and its implementation details entirely from the application. The interface wrapper allows applications to treat distributed data structures or objects as local within the application space. This enhanced interoperability by making the network transparent to the application developers.

The interface wrappers are generated from an extension of a prototype description language called Prototyping System Description Language (PSDL). The extended Description language (PSDL-ext) expands property definitions that are specific only to AICG model. Some of the salient features of the AICG model are:

- Distributed objects are treated as local objects within the application process. The application code needs not depend on how the object is distributed, since the local object copy is always synchronous with the distributed copy. (see section 5)
- Synchronization with various applications is automatically handled. Since the AICG model is based on the space transaction secure model, all operations are atomic. Deadlock is prevented automatically within the interface by having only a single distributed copy, and through transaction control. (see section 6, 8)

76

- Any type of object can be shared as long as the object is serializable. Any data structure and object can be distributed as long as it obeys and implements the java serializable feature (see section 10.2).
- Every distributed object has a lifetime. The distributed object lifetime is a period of time guaranteed by the AICG model for storage and distribution of the object. The time can be set by developer (see section 7).
- All write operations are transaction secure by default. AICG transactions are based on the Atomicity, Consistency, Isolation, and Durability (ACID) properties (see section 8).
- Clients can be informed of changes to the distributed object through the AICG event model (see section 9). A client application can subscribe for change notification, and when the distributed object is modified, a separate thread is spawned to execute the callback method defined by the developer.
- The wrapper codes are generated from high-level descriptive languages; hence, they are more manageable and more maintainable.

4. Developing Distributed Application with the AICG Tool

This section describes the steps for developing distributed applications using the AICG model. An example of a C4ISR application is introduced in section 4.2 to aid the explanation of the process. The same example will be used throughout this paper.

4.1 Development Process

The developer starts the development process by defining shared objects using the Prototyping System Description Language (PSDL). The PSDL is processed through a code generator (PSDLtoSpace) to produce a set of interface wrapper codes (figure 3). The interface wrapper contains the necessary codes for interaction between application and the space without the need for the developers to be concerned with the writing and removing of objects in the space. The developers can treat shared or distributed objects as local objects, where synchronization and distribution is automatically handled by the interface codes. The complete cycle for generating the interface codes is shown in figure 4.

PSDL definition of the distributed objects
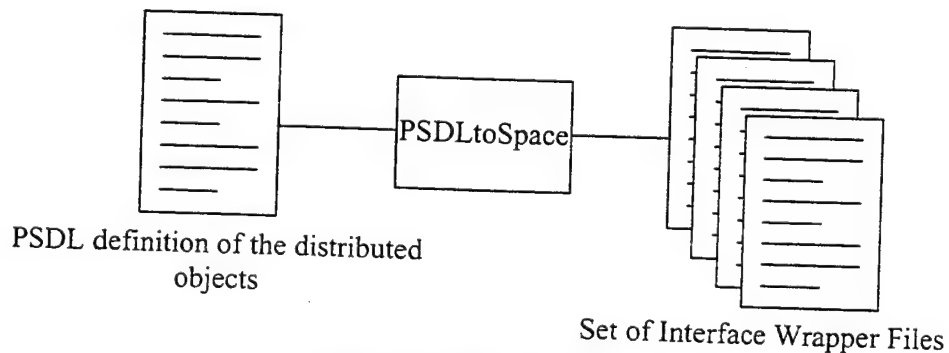
Set of Interface Wrapper Files

Figure 3, PSDL to Space
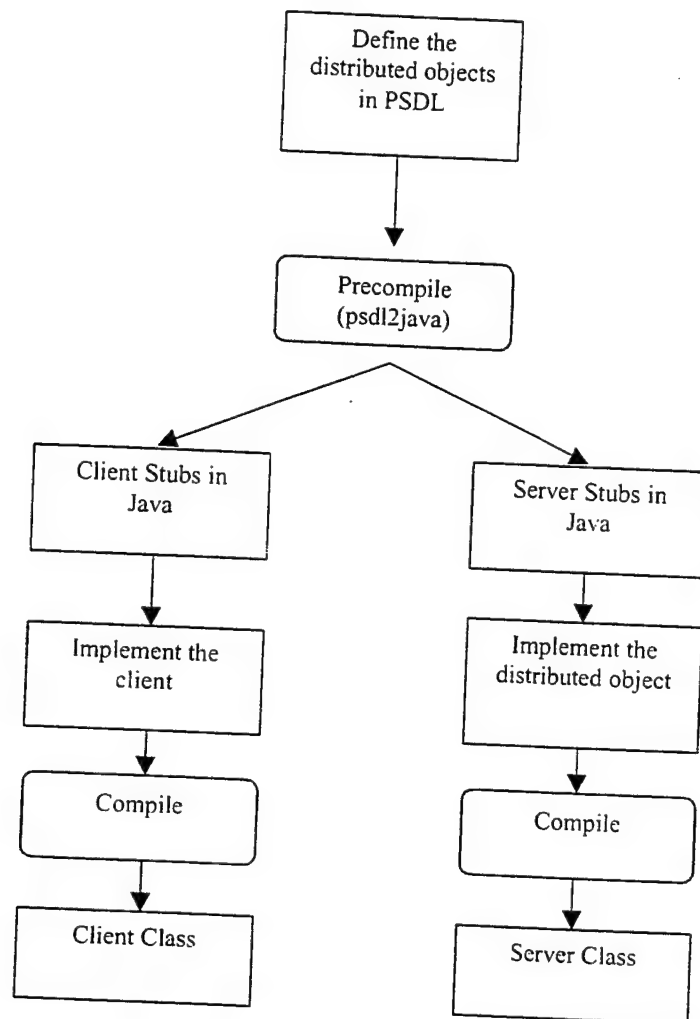
Figure 4, Generating the interface

4.2 Input definition to the Code generator

The following example demonstrates the development of one of the many distributed objects in the C4ISR system. Airplane positions picked up from the sensors are processed to produce track objects. These objects are distributed over a large network and used by several clients' stations for displaying the positions of planes. Each track or plane is identified by track number. The tracks are 'owned' by a group of track servers, and only the track servers can update the track positions and its attributes. The clients only have read access on the track data. Figure 5 shows the PSDL codes for the track object and its methods. Figure 6 shows the PSDL codes for the **Track_list** object and its methods.

```
Type track
 SPECIFICATION
  tracknumber: integer
 END

  OPERATOR track
  SPECIFICATION
   INPUT x:integer
  END
  IMPLEMENTATION
   SPACE
    PROPERTY SPACEMODE=
CONSTRUCTOR
   END
  END

  OPERATOR getID
  SPECIFICATION
   OUTPUT x:integer
  END
  IMPLEMENTATION
   SPACE
    PROPERTY SPACEMODE= READ
   END
  END

 OPERATOR setCallsign
  SPECIFICATION
   INPUT sign: string
  END
  IMPLEMENTATION
   SPACE
    PROPERTY SPACEMODE= WRITE
    PROPERTY TRANSACTIONTIME = 300
  END
 END

 OPERATOR getCallsign
  SPECIFICATION
   OUTPUT sign: string
  END
```

```
IMPLEMENTATION
   SPACE
    PROPERTY SPACEMODE= READ

  END
 END

  OPERATOR setPosition
  SPECIFICATION
   INPUT post : position_type
  END

  IMPLEMENTATION
   SPACE
    PROPERTY SPACEMODE = WRITE
    PROPERTY TRANSACTIONTIME = 2000

 END
  END

  OPERATOR getPosition
  SPECIFICATION
   OUTPUT post : position_type
  END
  IMPLEMENTATION
   SPACE
    PROPERTY SPACEMODE = READ

  END

 IMPLEMENTATION
  SPACE
   PROPERTY SPACENAME= DODSpaces
   PROPERTY OWNERSHIP = YES
   PROPERTY SECURITY = SERVER
   PROPERTY LEASE  = 12000
   PROPERTY CLONE = MANY
   PROPERTY NOTIFY = NO
   PROPERTY RETRY = 10
 END
```

Figure 5, Track example in PSDL

```
TYPE track_list
  SPECIFICATIO
  END

    OPERATOR track_list
    SPECIFICATION
    END
    IMPLEMENTATION
      SPACE
        PROPERTY SPACEMODE=
CONSTRUCTOR
      END
    END

    OPERATOR getID
    SPECIFICATION
      INPUT index: integer
      OUTPUT x:integer
    END
    IMPLEMENTATION
    SPACE
      PROPERTY SPACEMODE= READ

    END
  END

    OPERATOR setNewID
    SPECIFICATION
      INPUT id: integer
    END
```

```
  IMPLEMENTATION
    SPACE
      PROPERTY SPACEMODE= WRITE
      END
  END

  OPERATOR removeID
    SPECIFICATION
      INPUT id: integer
    END
    IMPLEMENTATION
      SPACE
        PROPERTY SPACEMODE= WRITE
        PROPERTY TRANSACTIONTIME = 2000
    END
  END
  END

IMPLEMENTATION
  SPACE
    PROPERTY SPACENAME= DODSpaces
    PROPERTY OWNERSHIP = YES
    PROPERTY SECURITY = SERVER
    PROPERTY LEASE  = 0
    PROPERTY CLONE = ONE
    PROPERTY NOTIFY = YES
    PROPERTY RETRY = 5
  END
```

Figure 6, Track list example in PSDL

The PSDL grammar used for the AICG is an extended version of the original PSDL grammar (Appendix A). PSDL model is very extensive and can be used to model an entire distributed system. However, the AICG only used a portion of the PSDL to describe the interface between systems. In another word, interactions between applications are defined using the PSDL but not the application itself. Because of this, slight modifications on the PSDL grammar were needed. The complete listing of the changes in the grammar statements can also be found in Appendix A.

The track PSDL starts with the definition of a *type* called **track**. It has only one identification field **tracknumber**. Of course, the **track** objects can have more than one field, but only one field is in this case is used to uniquely identify any particular **track** object. The type **track_list** shown in figure 5, on the other hand, does not need an identification field since there is only one **track_list** object in the whole system. **Track_list** is used to keep a list of all the active tracks **tracknumber** in the system at that moment in time.

All the operators (methods) of the *type* are defined immediately after the specification. Each method has a list of *input* and *output* parameters that define the arguments of the method.

The most important portion in the method declaration is the *implementation*. The developer must be able to define the type of operation the method supposed to perform. The operations are *constructor* (used to initialize the class), *read* (no modification to any field in the class) and *write* (modification is done to one or more fields in the class). These are necessary, as the code generated will encapsulate the synchronization of the distributed objects.

The other field in the implementation portion of the method, is *transactiontime*. *transactiontime* defines the upper limit in milliseconds within which the operation must be completed. The transaction property is discussed in detail in Section 8.

Upon running the example on figure 5 through the generator tool, a set of Java interface wrapper files are produced. Developers can ignore most of the generated files except the following:

- Track.java: this file contains the skeleton of the fields and the methods of the track class. The user is supposed to fill the body of the methods.
- TrackExtClient.java: this is the wrapper class that the client initialized and used instead of the track class.
- TrackExtServer.java: this is the wrapper class that the server initialized and used in replace for the track class.
- NotifyAICG.java : this class must be extended or implemented by the application if event-notification and call-back are needed.

The methods found in the trackExtClient and trackExtServer have the same method names and signatures of the track class. In fact, the track class methods are been called within trackExtClient or trackExtServer.

5. Distributed Data Structure and Loosely Coupled Programming

Conceptually a distributed data structure is one that can be accessed and manipulated by multiple processes at the same time without regard for which machine is executing those processes. In most distributed computing models, distributed data structures are hard to achieve. Message passing and remote method invocation systems provide a good example of the difficulty. Most of the systems tend to keep data structure behind one central manager process, and processes that want to perform work on the data structure must "wait in line" to ask the manager process to access or alter a piece of data on their behalf. Attempts to parallelize or distribute a computation across more than one machine face bottlenecks since data are tightly coupled by the one manager process. True concurrent access is rarely achievable.

Distributed data structures provide an entirely different approach where we uncouple the data from any particular process. Instead of hiding data structure behind a manager process, we represent data structures as collections of objects that can be independently and concurrently accessed and altered by remote processes. Distributed data structures allow processes to work on the data without having to wait in line if there are no serialization issues.

The distributed protocol for modification ensures synchronization by enforcing that a process wishing to modify the object has to physically remove it from the space, alter it and write it back to the space. There can be no way for more than one process to modify an object at the same time. However, this does not prevent other processes from overwriting the corrected data. For example, in the normal JavaSpace, process A instead of performing a "take" follow by a "write operation, the programmer wrote a "read" operation, followed by a "write" operation. This results in 2 copies of the object in the Space. The AICG model prevents this by encapsulating the 3 basic commands from the developers. All modification on the object are automatically translated to "take", followed by "write" and all operations that access the fields of the distributed object are translated to "read". These ensure that local data are up-to-date and serialization is maintained.

Loosely-coupled programming has it pitfalls also. Distributed objects may be lost if a process removes it from the space and subsequently crashes or is cut off from the network. Similarly, the system may enter in a deadlock state if processes request more than one distributed object while, at the same time, holding on to distributed objects required by other processes. In cases like this, the AICG model groups multiple operations into a transaction to ensure that either all operations complete or none occur, thereby maintaining the integrity of the application. With transaction control, deadlock is prevented if the process did not complete the operation within a certain permitted time. The application can retry the operation immediately or wait for a random time before performing the operation again

6. Synchronization

Synchronization plays a crucial role in any design of distributed application. Inevitably, processes in a distributed system need to coordinate with one another and avoid bringing the system into an unstable state such as deadlock. Creating distributed applications with AICG can significantly ease the burden of process synchronization since synchronization is already built into the AICG operations. Multiple processes can read an object in a space at any time, but when a process wants to update an object, it has to remove it from the space and thereby gain exclusive access to it first. Hence, coordinated access to objects is enforced by the AICG interface doing *read*, *take* and *write* operations.

More advanced and complex synchronization schemes can be easily build upon from the basic atomic features of the AIGC operations. An example is semaphores. Semaphores, a synchronization construct that was first used to solve concurrency problems in operating systems, are commonly found in multithreaded programming languages, but are more difficult to achieve in distributed systems. Semaphores are typically implemented as integer counters that require special language or hardware support to ensure the atomic properties of the *UP* (signal) and *DOWN* (wait) operations. Using AIGC space model, we could easily implement a semaphore as a shared variable that holds an integer counter. By assigning a distributed variable or object as a semaphore, groups of distributed objects can be synchronized. Hence, the AIGC model permits the developers to develop more complicated distributed applications without being concerned about synchronization and deadlock. Furthermore, all operations within the AICG model can impose transaction control with

timeout monitoring. After the timeout period, the transaction would rollback the application to a stable state.

7. Object Life Time (Leases/Timeout)

Leasing provides a methodology for controlling the life span of the distributed objects in the AICG space. This allows resources to be freed after a fixed period. This model is beneficial in the distributed environment, where partial failure can cause holders of resources to fail thereby disconnecting them from the resources before they can explicitly free them. In the absence of a leasing model, resources could grow without bound.

There are other constructive ways to harness the benefit of the leasing model besides using it as a garbage collector. As for example, in a real-time system, the value of the information regarding some distributed objects becomes useless after certain deadlines. Accessing obsolete information can be more damaging in this case. By setting the lease on the distributed object, the AICG model automatically removes the object once the lease expires or the deadline is reached.

Java Spaces allocate resources that are tied to leases. When a distributed object is written into a space, it is granted a lease that specifies a period for which the space guarantees its storage. The holder of the lease may renew or cancel the lease before it expires. If the leaseholder does neither, the lease simply expires, and the space removes the entry from its store.

The AICG model simplified the Java Space lease model into two configurations. These are
1. Generally, the distributed object lasts forever as long as the space exists, even if the leaseholder (the process that creates the object) has died. This configuration is enabled by setting the *SPACE lease* property in the Implementation to 0.
2. In real-time environment, the distributed object lasts for a fixed duration of x ms specified by the object designer. To keep the object alive, a write operation must be performed on the object before the lease expires. This configuration is set through the *SPACE lease* property in the Implementation to the time in ms required.

Hence, the developer must provide due consideration towards leasing while developing the application. If an object has a life time, it must be renewed before it expires. In the AICG model, renewal is done by calling any method that modifies the object. If no modification is required, the developer can consider defining a dummy method with the spacemode set to "write". Invoking that method will automatically renew the lease.

8. Transactions
The AICG model uses the Jini Transaction model, which provides generic services concerning transaction processing in distributed computing environment.

8.1 Jini Transaction model:

All transactions are overseen by a transaction manager. When a distributed application needs operations to occur in a transaction secure manner, the process asks the transaction manager to create a transaction. Once a transaction has been created, one or more processes can perform operations under the transaction. A transaction can complete in two ways. If a transaction commits successfully, then all operations performed under it are complete. However, if problems arise, then the transaction is aborted and none of the operations occurs. These semantics are provided by a two-phase commit protocol that is performed by the transaction manager as it interacts with the transaction participants.

8.2 AICG Transaction model

AICG model encapsulates and manages the transaction procedures. All operations on the distributed object can be either with transaction control or without. Transaction control operations are controlled with a default lease of 2 sec. This default value of leasing time may, however, be overriden by the user. This is kept by the transaction manager as a leased resource, and when the lease expires before the operation committed, the transaction manager aborts the transaction.

Transactions have the following desirable effect on the semantics of the AICG operations. When a distributed object is created, the object is not seen or accessible outside of the transaction until the transaction commits. However, when a distributed object is updated or read under transaction, it can come from new object created within the transaction or objects in the space.

The AICG model by default, enable all transaction for *write* operations and the transaction lease time is two seconds. The developer can modify the lease time through the PSDL SPACE *transactiontime* property.

PROPERTY
transactiontime= 0: Disable transaction for that method
/n: Set the lease time to n ms.

All the read operations in the AICG model do not have transactions enabled. However, the user can enable it by using the property transactiontime with the upper limit in transaction time for the read operation. To used the same transaction for more than one operation, the following property must be set.

PROPERTY
transactionID = 99 : An ID number that are the same for more than one method.

9. AICG Event Notification

In the distributed and loosely-coupled programming environment, it is desirable for an application to react to changes or arrival of newly distributed objects instead of "busy waiting" for it through polling. AICG provides this feature by introducing a callback mechanism that invokes user-defined methods when certain conditions are met.

Java provides a simple but powerful event model based on event sources, event listeners and event objects. An event source is any object that "fires" an event, usually based on some internal state change in the object. In this case, writing an object into space would generate an event. An event listener is an object that listens for events fired by an event source. Typically, an event source provides a method whereby listeners can request to be added to a list of listeners. Whenever an event source fires an event, it notifies each of its registered listeners by calling a method on the listener object and passing it an event object.

Within a Java Virtual machine (JVM), an application is guaranteed that it will not miss an event fired from within. Distributed events on the other hand, had to travel either, from one JVM to another JVM within a machine or between machines networked together. Events traveling from one JVM to another may be lost in transit, or may never reach their event listener. Likewise, an event may reach its listener more than once.

Space-based distributed events are built on top of the Jini Distributed Event model, and the AICG event model further extends it. When using the AICG event model, the space is an event source that fires events when entries are written into the space matching a certain template an application is interested in. When the event fires, the space sends a remote event object to the listener. The event listener codes are found in one of the generated AICG interface wrapper files. Upon receiving an event, the listener would spawn a new thread to process the event and invoke the application callback method. This allows the application codes to be executed without involving the developer in the process of event-management.

There are a few steps for setting up AICG event for a particular application. Firstly, the distributed objects must have the SPACE properties for *Notification* set to yes. One of the application classes must *implement* (java term for inherit) the notifyAICG abstract class. The notifyAICG class has only one method, which is the callback method. The user class must override this method with the codes that need to be executed when an event fires.

10. AICG Design

This section explains the design of the AICG and the codes that are generated from psdl2java program. The codes used in this section to explain the AICG and the development processes are generated from the track PSDL of section 4.2.

10.1 AICG Architecture

The AICG architecture consists of four main modules. They are the Interface modules, the Event modules, Transaction modules and the Exception module. The interface modules

implement the distributed object methods and communicate directly with the application. In reference to the example in section 4, the interface modules are entryAICG, track, trackExt, trackExtClient, trackExtServer. Instead of creating the actual object (track), the application should instantiate the interface object either the trackExtClient or trackExtServer. Event modules (eventAICGID, evenAICGHandler, notifyAICG) handle external events generated from the JavaSpace that are of interest to the application. Transaction modules (transactionAICG, transactionManagerAICG) support the interface module with transaction services. Lastly, the exception module (exceptionAICG) defines the possible types of exceptions that can be raised and need to be catch by the application. Figure 7 below shows the architecture of the generated interface wrapper and the interaction with the other modules and application.

Each time the application instantiate a track class by creating a new trackExtServer, the following events take place in the Interface:

1. An Entry object is created together with the track object by the trackExtServer. The tack object is placed into the Entry object and stored in the space.
2. Transaction Manager is enabled.
3. The reference pointer to trackExtServer is returned to the application.
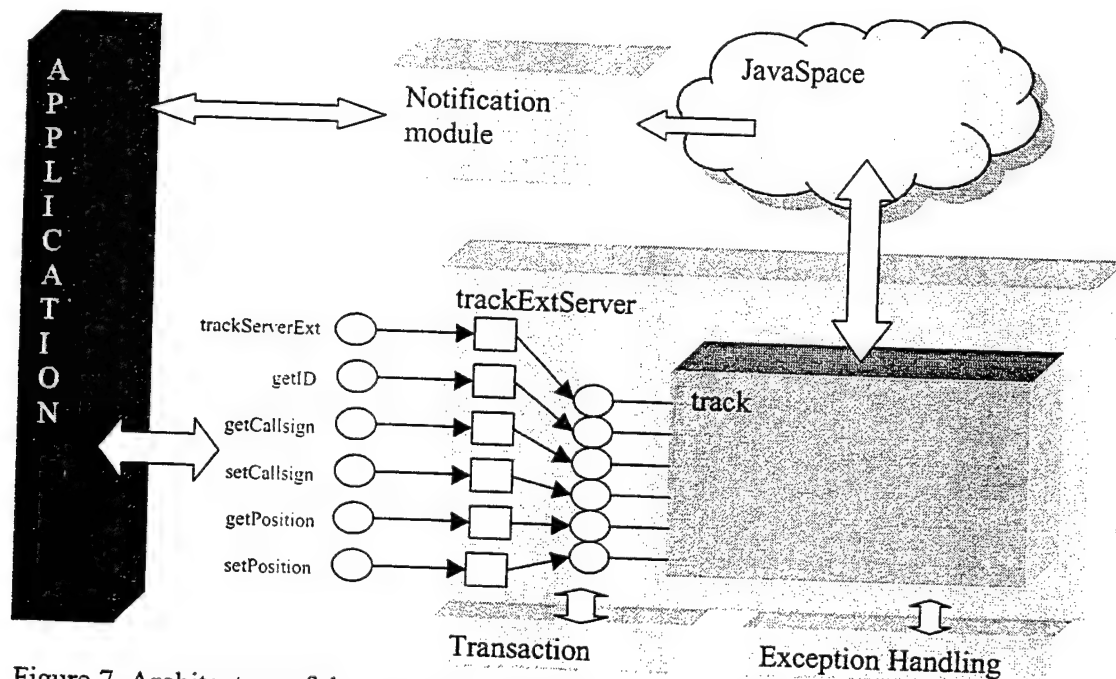

Figure 7, Architecture of the generated interface wrapper and the interaction with the other modules and application

Each time a method (getID, getCallsign, getPosition) that does not modify the contents of the object is invoked, the following events take place in the Interface:

1. When the application invokes the method through the Interface (trackExtServer/trackExtClient).

2. The Interface performs a Space "get" operation to update the local copy.
3. The method is then executed on the updated copy of the object to return the value back to the application.

Each time a method (setCallsign, setPosition), which does modify the contents of the object is invoked, the following events take place in the Interface:
1. When the application invokes the method through the Interface
2. The interface performs a Space "take" operation, which retrieves the object from the space.
3. The actual object method is then invoked to perform the modification.
4. Upon completion of the modification, the object is returned to the space by the interface using a "write" operation.

10.2 Interface Modules

The interface modules consist of the following modules; an entry (entryAICG) that are stored in space, the actual object (trackExt)that are shared and the object wrapper (trackExt, trackExtClient, trackExtServe.).

10.2.1 Entry

A space stores *entries*. An entry is a collection of typed objects that implements the Entry interface. The base class of the AICG distributed object:

```
public abstract class entryAICG implements Entry
    {
        // main identifcation number
        public Integer entryID;
        // required by JavaSpace //default constructor
        public entryAICG( )
        { }
        public entryAICG(int id){
            entryID = new Integer(id);
        }
        // return the object stored in   //the entry
    public abstract Object
        getObject( );
}
```

The Entry interface is empty; it has no methods that have to be implemented. Empty interfaces are often referred to as "marker" interfaces because they are used to mark a class as suitable for some role. That is exactly what the Entry interface is used for, to mark a class appropriate for use within a space.

All entries in the AICG extend from this base class. It has one main public attribute, an identifier and an abstract method that returns the object. Any type of object can be stored in the entry. The only limitation is that the object must be serializable. Serializable allows the java virtual machine to pass the entire object by value instead of by reference. Here is an example "track" entry codes generated by the AICG from the PSDL file in figure 4. The interface contains the object track in one of the field and an ID.

```
public abstract class trackEntry
    extends entryAICG
{
// id is required if there are more
// than one similar object in
// the space
    public Integer id;
    // track object
    public track data;
    // default Constructor
     public trackEntry(){ }
    // Constructor with information
    //extracted from the track PSDL
    // file.
    public trackEntry(int aid, Integer
        inID, track inData){
            super(aid);
            data = inData;
            id = inID;
    }
    public Object getObject(){
       return data;
    }
}
```

All Entry attributes are declared as publicly accessible. Although it not typical of fields to be defined in public in object-oriented programming style, the associative lookup is the way the space-based programs locate entries in the space. To locate an object in space, a template is specified that matches the contents of the fields. By declaring entry fields public, it allows the space to compare and locate the object. AICG encourage object-oriented programming style by encapsulating the actual data object into the entry. The object attributes can then be declared as private and made accessible only through clearly defined public methods of the object.

10.2.2 Serialization

Each distributed interface object is a local object that acts as a proxy to the remote space object. It is not a reference to a remote object but instead a connection passes all operations and value through the proxy to the remote space. All the objects must be serializable in order to meet this objective. The Serializable interface is "marker" interface that contains no methods and serves only to mark a class as appropriate for serialization. Here is the Serializable interface:

```
public abstract interface Serializable
{
    // this interface is empty
}
```

In that case, the *track* class of the example needs to implement the interface Serializable.
```
public class track implements
    Serializable {
    .  .
    // since Serializable is a marker
    // interface no methods need to be
    //override.
}
```

10.2.3 The Actual Object

We now look at the actual objects that are shared between the servers and clients. The psdl2java generates a skeleton version of the actual class with the methods names and its arguments. The body of the methods and its fields need to be filled by the developers. The track class generated is shown below:

```
public class track implements
java.io.Serializable
{
    private Integer trackNumber;

    public track(int inID){
    // insert the body here
    }
    public int getID(){
     // insert the body here
    }

    public void setPosition
        (position_type post){
     // insert the body here
    }

    public position_type getPosition(){
    // insert the body here
    }

    public String getCallsign(){
    // insert the body here

    }
    public void setCallsign(String
        sign){
    // insert the body here
    }

    // automatically generated do
    // not delete!!
    public Integer autoGetID1(){
        return trackNumber ;

    }
}
```

10.2.4 Object Wrapper

Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf (COTS) software products that require no modification of those products [1]. It consists of two parts, an adapter that provides some additional functionality for an application program at key external interfaces, and an encapsulation mechanism that binds the adapter to the application and protects the combined components [1].

In this context, the software being protected contains the actual distributed objects, and the AICG model has no way of knowing the behaviors of the distributed object other than the type of operations of the methods. The adapter intercepts all invocations to provide additional functionalities such as synchronization between the local and distributed object, transaction control, events monitoring and exceptions handling. The encapsulation mechanism has been explained in the earlier section (AICG Architecture). Instead of instantiation of the actual

89

object, the respective interface wrapper is instantiated. Instantiating the interface wrapper would indirectly instantiate the actual object as well as storing the object in the space.

Three classes generated for every distributed object. There are named with the object name appended with the following Ext, ExtClient, and ExtServer.

10.3 Event Modules

The event modules consist of the event callback template (notifyAICG), the event handler (evenAICGHandler) and the event identification object (eventAICGID).

10.3.1 Event Identification object

The event identification object is used to distinguish one event from others. When an event of interest is registered, an event identification object is created to store the identification and event source. Together these two properties act to uniquely identify the event registration.

The object has only two methods, an 'equals' method that check if two event identification objects are the same and a 'to string' method which is used by the event handler for searching the right event objects from the hash table.

10.3.2 Event Handler

Event Handler is the main body of the event operation in the AICG model. It handles registration of new events, deletion of old events, listening for event and invoking the right callback for that event. Inside the event handler are in fact, three inner classes to perform the above functions. Events are stored in a hash table with the event identification object as the key to the hash table. This allows fast retriever of the event object and the callback methods.

The event handler listens for new events from the space or other sources. When an object is written to the space, an event is created by the space and captured by the all the listeners. The event handler would immediately spawn a new thread and check whether the event is of interest to the application.

```
// call when an external event is
// "fired".
public void run() {
    Object source = event.getSource();
    long id = event.getID();
    long seqN =
        event.getSequenceNumber();
    // create a new event identifcation
    //object
    eventAICGID keyID= new
        eventAICGID(id,source);
    registerAICG tempReg;

    String key = new
        String(keyID.toString());
    // check if the key exist in the
    // hash table (storage)
    if ((tempReg = (registerAICG)
        storage.get(key)) !=null)
```

```
            {
            // check if the event is an old or
            // duplicate event
               if (seqN > tempReg.seqNum) {
                  tempReg.seqNum = seqN;
                  src.listenerAICGEvents
                     (tempReg.anyObj);
               } else {
               // old events ignored
                  return;
               }
            }
         }
}// end of notifyHandler
```

10.3.3 The Callback Template

The callback template is a simple interface class with an abstract method listenerAICGEvents. Its main function is to allow the AICG model to invoke the application program when certain events of interest is "fired". As explain earlier, the template need to be implemented by the application that wishes to have notification.

```
public interface notifyAICG
{
   public abstract void
   listenerAICGEvents(Object obj);

}
```

10.4 The Transaction Modules

The transaction modules consist of transaction interface (transactionAICG) and the transaction factory (transactionManagerAICG).

The transaction interface is a group of static methods that are used for obtaining reference to the transaction manager server somewhere on the network. It uses the Java RMI registry or the look-up server to locate the transaction server.

The transaction factory uses the transaction interface to obtain the reference to the server, which is then used to create the default transaction or user-define transaction. In short the transaction factory can perform the following:

1. Invoke the transaction interface to obtain a transaction manager.
2. Create a default transaction with lease time of 5 seconds.
3. Create a transaction with a user define lease time.

10.5 The Exception Module

The exception module defines all the exception code that is return to the application when certain unexpected conditions occur in the AICG model. The exception include
- "NotDefinedExceptionCode"; unknown error occur.
- "SystemExceptionCode"; system level exceptions, such disk failure, network failure.
- "ObjectNotFoundException"; the space does not contain the object.

- "TransactionException"; transaction server not found, transaction expire before commit.
- "LeaseExpireException"; object lease has expired.
- "CommunicationException"; space communication errors.
- "UnusableObjectException"; object corrupted.
- "ObjectExistException"; there another object with the same key in the space.
- "NotificationException"; events notification errors.

11. Conclusion

This paper demonstrates the ease of sharing distributed objects and automates the generation of generic interface wrappers directly from the Prototype System Description Languages. However, the design has a performance price penalty. Every read operation requires the interface to synchronize the local object with the distributed object before the value is returned. Every write operation requires two Space operations. Adding the overhead for transactions, event monitoring and control, reading operations are in the range of a hundred milliseconds and writing is in the range of a few hundred of milliseconds. The high overhead lies within the Java Virtual Machine (JVM), the JavaSpace Model and the network latency. Current versions of JVM and JavaSpace are in a premature state in terms of performance. Even so, the performances are still suitable for most applications that are not time critical. Similar implementations of distributed systems with the above features of AICG interface in CORBA and Java would not perform any better. Hence, the AICG model is still a viable option in developing interface wrapper for distributed system.

12. References

[1] Naval Postgraduate School Report NPSCS-00-001, *Interoperability Technology Assessment for Joint C4ISR Systems*, by Valdis Berzins, Luqi, Bruce Shultes, Jiang Guo, Jim Allen, Ngom Cheng, Karen Gee, Tom Nguyen, and Eric Stierna , September 1999.

[2] Nicholas Carriero, and David Gelernter, *"How to Write Parallel Programs: A Guide to the Perplexed"* ACM Computing Surveys, September 1989, pp.102-122.

[3] David Gelernter, *"Generative Communication in Linda"*, ACM Transaction on Programming Languages and Systems, Vol. 7, No. 1, January 1985, pp. 80-112.

[4] Bill Joy, *The Jini Specification*, Addison Wesley, Inc., 1999.

[5] Edward Keith, *Core Jini*, Prentice Hall, PTR, 1999.

[6] Eun-Gyung Kim, *"A Study on Developing a Distributed Problem Solving System"* IEEE Software, January 1995, pp. 122-127.

[7] Fred Kuhns, Carlos O'Ryan, Douglas Schmidt, Ossama Othman, and Jeff Parsons, *"The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware"*, IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN' 99), August 25-27, 1999.

[8] David Levein, Sergio Flores-Gaitan, and Douglas Schmidt, *"An Empirical Evaulation of OD Endsystem Support for Real-time CORBA Object Request Brokers"*, Multimedia Computing and Network 2000, January 2000.

[9] Luqi, and Valdis Berzins, *"Rapidly Prototyping Real-Time Systems"*, IEEE Software, September 1988, pp. 25-35.

[10] Naval Postgraduate School, *A Proposed Design for a Rapid Prototyping Language*, by Luqi, Valdis Berzins, Bernd Kraemer, and Laura White, March 1989

[11] Luqi, Mantak Shing, "*CAPS – A Tool for Real-Time System Devleopment and Acquisition*", Naval Research Review, Vol 1, 1992, pp.12-16.

[12] Luqi, Valdis Berzins, and Raymond Yeh, "*A Prototyping Language for Real-Time Software*", IEEE Software, October 98, pp.1409-1423.

[13] Kevin Sullivan, Mark Marchukov, and John Socha, "*Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model*" IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August 1999, pp. 584-599.

[14] Antoni Wolski, "*LINDA: A System for Loosely Integrated DataBases*", IEEE Software, January 1989, pp. 66-73.

[15] Andrew Xu, and Barbara Liskov, "*A Design of a Fault-Tolerant, Distributed Implementation of Linda*", IEEE Software January 1989, pp. 199-206.

[16] Jingshuang Yang, and Gail Kaiser, "*JPernLite: Extensible Transaction Services for the WWW*", IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 4, July/August 1999, pp. 639-657.

Computer Aided Prototyping in a Distributed Environment

Jun Ge, Valdis Berzins, Luqi
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943
U.S.A.

Abstract[+]

Previous work on computer-aided prototyping system (CAPS) is stepping into a distributed environment to meet the requirement of integrating legacy systems in heterogeneous network. A three-module architecture design, including Supporting Database, System Tools and Execution Manager, is proposed in this paper for the distributed CAPS system (DCAPS). By using wrapper/glue technique, different prototyping tools in a heterogeneous environment share the input/output data files for prototypes. The architecture is generalized for the communication among legacy systems for data interchange. DCAPS not only provides a useful tool for distributed real-time system prototyping, but also is a demonstration of distributed system in heterogeneous environment.

Key words: software interoperability, fast prototyping, distributed system, multi-agent system

1. Introduction

Computer aided prototyping has been found useful in software development, especially for large real-time systems. Prototyping provides the capability to accurately simulate requirements in new application areas. Previous work such as the Computer Aided Prototyping System (CAPS) has demonstrated real-time issues, software reuse and process scheduling in fast prototyping for a single processor computing environment [1-3]. However, it is still hard to make use of existing systems in a distributed environment, especially for real-time systems under a heterogeneous environment. With the fast development of networks and the Internet, interoperability has become the focus of current research. This paper extends research on CAPS to distributed and network computing.

Distributed real-time software system prototyping and interoperability in a heterogeneous environment form the focus of this paper. In recent years, hard real-time, soft real-time and embedded systems are increasingly important in various application areas from e-business to military applications. These systems have strict requirements on accuracy, safety and reliability. Usually such software is large and built on several legacy systems to make use of the partial or full functionalities of these legacy systems. When the legacy systems are physically located in a distributed network, they are connected through certain network protocols. Fast prototyping of these systems helps the users in analysis, design, implementation, verification, validation and optimization. Approaches for modeling, realizing, reconfiguring and allocating logical processes and interactions to processors and communication links are needed to make prototyping useful in this domain.

This paper describes a distributed CAPS system (DCAPS) to fulfill the requirements for distributed software prototyping. Prototype System Description Language (PSDL), a prototyping language, is applied in the description of the real-time software in DCAPS system. PSDL provides the specifications not only for real-time constraints, but also for the connection and interaction among software components. PSDL has open syntax for the design of new features that arise in the context of distributed computing. Wrapper and glue technology is applied for the normalization and data transfer of legacy systems. A multi-agent technique is used to manage the execution process.

Section 2 introduces the three-module architecture of DCAPS system. All the modules are described in detail in Section 3, 4 and 5 separately. Section 6 gives a simple example prototype in DCAPS.

2. System architecture

Earlier work on computer-aided prototyping system (CAPS) uses PSDL, a prototype description language, to describe the real-time software [4]. PSDL itself has an open structure so that the user is able to define new properties for software components, such as new-added network configurations. CAPS prototypes a software system in the following steps. First, user selects the software components from the reusable component libraries b construct the prototype in a graphic editor. This prototype is saved as a plain text file in PSDL format. User may also use the graphic user interface (GUI)

generator provided by CAPS to create the new GUI interface for the prototype. Then, the translator and scheduler work on this PSDL file to generate the wrapper/glue code and dynamic/static schedules respectively. Both the source code of reusable components and automatic generated source code will be compiled together to get the executable final software. It will be tested in CAPS (simulation) for both the execution correctness and the real-time requirements.

As described above, CAPS consists of various prototyping tools to provide all these functionalities. They play different roles during the prototyping process. For example, the scheduler just needs the information of timing constraints for every component, while the translator does not care about such information other than the network configurations and data type definitions. When new properties are enabled in PSDL description of the prototype, for instance to prototype a networked software, some tools must be updated by new generations while the rest stay the same. Therefore, the architecture of CAPS must consider the evolution of its own components.

CAPS tools were originally developed in SunOS operating system for components which are located on one processor. To consider the user's requirement, the user interface is required to migrate to Windows NT operating system. At the same time, the old operating system is not supported by some new technologies. To avoid the complexity of migrating the whole system to a new operating system, CAPS now has to work in a distributed and heterogeneous environment. A new architecture becomes important for the system. On the other hand, CAPS is required to prototype software systems in distributed and heterogeneous environments. The requirements to develop the distributed CAPS (DCAPS) are consistent for constructing the distributed software prototypes, i.e., DCAPS itself is a demonstration of distributed software construction. A three-module architecture is proposed to design the distributed CAPS system (DCAPS).

From the viewpoint of prototyping procedure, DCAPS can group its tools into three basic modules (Figure 1).
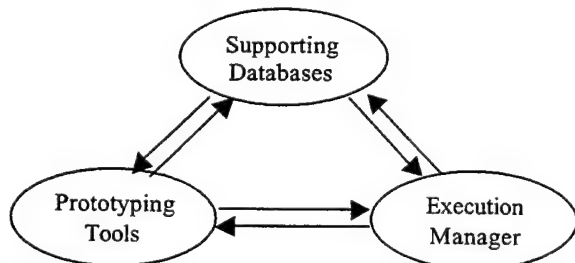

Figure 1. Three-module architecture design of DCAPS

In this architecture, DCAPS provides users support from three aspects. Databases help users to manage and reuse the prototyping requirements and reusable software components. It also validates the prototypes for components' evolution. Prototyping tools help user in automatically generating connection code, GUI code, and data type conversion code among components during the design process. Execution manager controls and visualizes the simulation process to validate the system design, particularly on real-time constraints.

DCAPS inherits prototyping tools that were implemented in different operating systems including SunOS, Solaris and Windows NT. It provides different user interfaces for multiple operating systems including Windows NT. All the tools, which are in the three modules, are located in a distributed environment during one prototyping job.

3. Supporting databases

Supporting databases provide intelligent guidance to users so that in a form of adaptive control it is integrated into the system prototyping. There are two types of database support involved in DCAPS system. One is the software reuse database. It contains the specifications for all the reusable software components so that they are able to be retrieved and to be accessed during the prototyping procedure and the execution (simulation). Software version control should also be considered within this database support. The other is the requirement database. It allows users to reuse the previous prototypes that are stored in the database. Thus it may shorten the design cycle and even optimize the design. The decomposition of this module is shown in Figure 2.
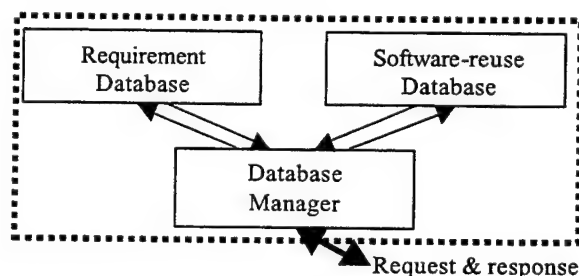

Figure 2. Supporting database system

The browse and retrieve operations for the database includes both syntactic exclusion and semantic exclusion to narrow the search range [5][11].

4. Prototyping tools

Prototyping tools module is decomposed as follows (Figure 3). It includes GUI for various operating systems,

which includes a PSDL graphic editor, the prototype scheduler [9], the prototype translator (automatic code generator for data communication among components), source code compilers and code optimizers for various languages and operating systems. The major operating systems considered in DCAPS are SunOS, Solaris and Windows NT. Job Dispatcher works on a server platform to receive user's commands from GUI and to dispatch jobs to correspondent tools.

The compiler in different operating systems just needs to work with the correspondent automatically generated code. With the change of language in a specific operating system, it is not necessary to change the other components of DCAPS.
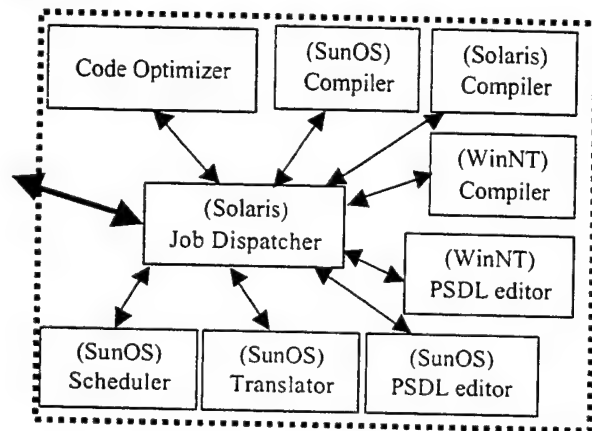

Figure 3. Decomposition of System Tools

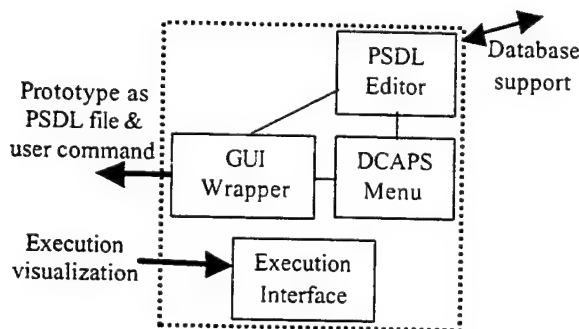The DCAPS GUI can be further decomposed as in Figure 4.

Figure 4. Decomposition of DCAPS GUI

The graphic PSDL editor should be enhanced for new-added properties in the PSDL description of prototype, such as network configuration, different timing constraint, etc. Even in such cases, the system architecture does not have to change at all except that the respective modules are replaced.

The different tools, which are located in different computers, communicate with each other through TCP/IP protocol. The wrapper/glue technique is applied. However, because the data types in communication are known to each other, the wrappers among different tools are blank to each other.

5. Execution manager

The execution of the distributed system, i.e., the simulation of the prototype, is managed by the Execution Manager. It uses a virtual centralized synchronization timer for different task schedules in different processors. This subsystem must compensate for clock drift due to differences in clock rates without violating global timing constraints as long as clock drift rates remain within specified bounds. A multi-agent system is used in the distributed work to coordinate the computing processes.

The Prototyping Scheduler generates one specific task schedule (both dynamic and static) for each node. Execution Manager provides a centralized Executor to administrate and to synchronize the processes in different platforms on which reusable components are located (Figure 5). The procedure of execution is also sent back to GUI of DCAPS so that the user may see a visualized process and have clear information on the prototype.

Legend: ⊠ local timing agents
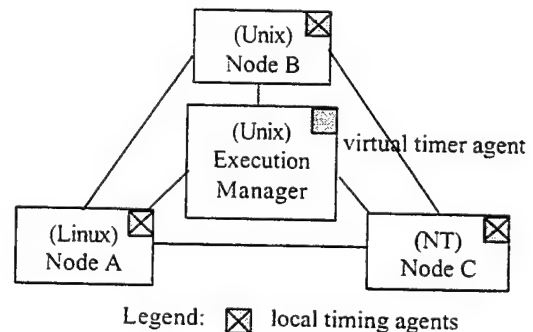
Figure 5. Execution model for a distributed system

In each node, for all the legacy components, the wrapper/glue technology is applied in data interchange (Figure 6). A form of software wrapper and glue technology provides standardized interactions between legacy systems in a heterogeneous network in DCAPS. It makes interoperability and integration possible for a distributed structure. Legacy systems under the wrappers collaborate through the message passing approach in the glue connection. Wrappers provide a generic interface for every single legacy system so that its input and output become uniform, both for consuming data from other legacy systems and for generating data to others. On the other hand, glue structure supports an abstract data class

96

for data transfer. It encodes any type of data to a common type before putting it into a data stream at the sender's end. At the receiver's end, the data is decoded to the required data type that may be different from that at the sending end. Wrapper and glue concepts are the basis of a formal model for software and hardware co-design.

A multiple-agent system is generated automatically by the Prototyping Translator tool in the architecture as the "glue" for the network communication of the legacy system's inputs and outputs. For each input/output data flow, an agent is associated as an automatic pipe of data transmission. It makes use of the run-time library of network communication according to the specific network protocol in the node that is provided in component information. This "glue" allows the legacy systems not to worry about the network settings for the communication to other components. The communication among agents can reference to several available techniques such as JavaSpace, Jini [7], etc. The technology used in real application should be selected according to the real network configuration.

The "wrapper" code works with the component for data type control/conversion, firing condition, exception handling, timing constraints, etc. The "wrapper" is simply composed in several different layers so that all the features that user concerns are tunable according to user's selections. The "wrapper" communicates to the agents for data outgoing and incoming. Under certain specific conditions, some layer of the wrapper may become transparent based on enhanced information. For example, in the design of DCAPS, the input/output of different prototyping tools are standardized in advance. Therefore, the data type conversion is not required. Because DCAPS itself does not have real-time constraint, the wrapper for timing constraints is transparent.
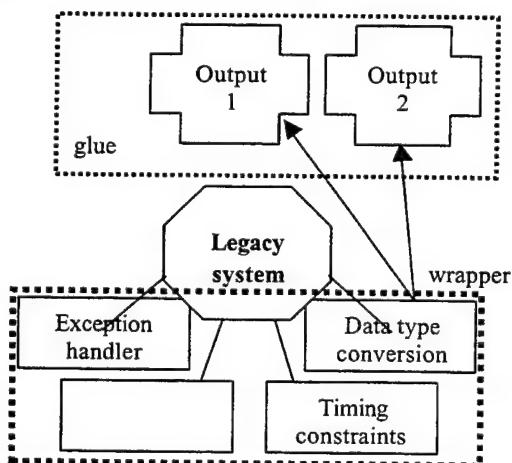
Figure 6. Wrapper/glue architecture for one component

For each processor, a local timing agent manages the execution tasks under the schedule. I/O data of each component is received/sent between legacy system and the uniform software wrapper, which is automatically generated and transferred through glue agents generated by glue code, which hides the specific network configurations via derived design and network mode/parameters.

6. Prototyping example

The system of a weather station is prototyped in DCAPS to demonstrate the ability of prototyping the distributed software in heterogeneous operating system.
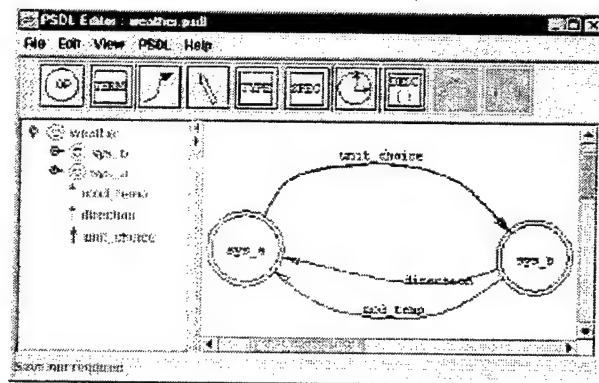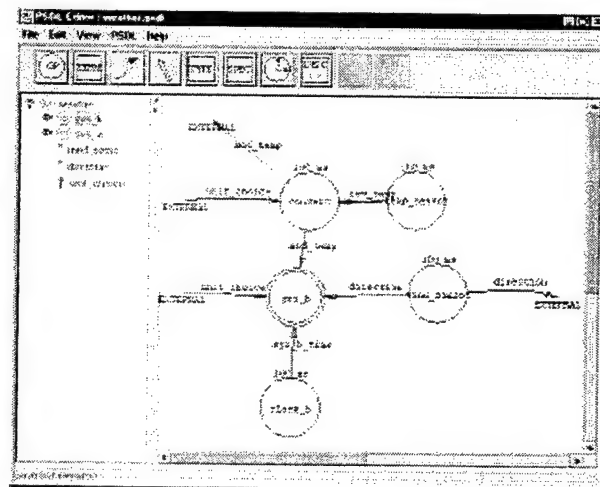

Figure 7. Top level of weather-station prototype
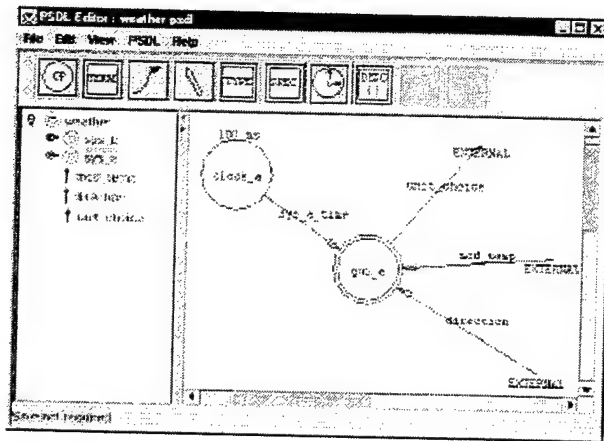

Figure 8. Decomposition of sys_b

97

Figure 9. Decomposition of sys_a

Figure 10. Properties configuration for components

As shown in Figure 7-9, weather station system consists of two parts: sys_b is the sensor and sys_a is the controller. The sensor system includes two sub-sensors which are wind direction sensor and temperature sensor. The measurements are converted in specified units. It reports the measurement results to the controller. The controller sends control signal of signal unit to the sensor system so that the sensor can be configured automatically. Both the sub-systems have their own user interfaces in the local systems.

The two sub-systems are located in different computers. They are connected through network in TCP/IP protocol. A SOCKET communication run-time library is provided for data interchange.

DCAPS provides the graphic user interface to edit the prototype in multi-level. For each component, it provides an interface (Figure 10) so that user may specify properties such as timing constraints, network configuration, data flow type, etc. PSDL editor also supports a GUI code generator so that user can create a personal-style user interface for the prototype.

7. Conclusions

The DCAPS system provides a useful tool for distributed real-time software fast prototyping. A three-module architecture is proposed to make DCAPS system suitable for distributed environment. The wrapper/glue method used in DCAPS can be generalized to system construction and interconnection of legacy systems. By automatically generating the codes for the "wrappers and glue" and providing a powerful environment, DCAPS allows the designers to concentrate on the difficult interoperability problems and issues, freeing them from implementation details. It also enables easy reconfiguration of software and network properties to explore design alternatives. DCAPS is an on-going research project for the development and refinement of its prototyping tools.

References

1. Luqi, V. Berzins, Rapidly prototyping real-time systems, IEEE Software, September 1988, pp. 25-36
2. Luqi, W. Royce, Status report: computer-aided prototyping, IEEE Software, Vol. 9, No. 6, November 1992, pp. 77-81
3. Luqi, M. Shing, Real-time scheduling for software prototyping, J. of Systems Integration, special issue on computer-aided prototyping (Vol. 6, No. 1, 1996), pp. 41-72
4. Luqi, V. Berzins, R. Yeh, A prototyping language for real time software, IEEE Transactions on Software Engineering, October 1988, Vol. 14, No. 10, pp. 1409-1423
5. R. Steigerwald, Luqi, J. McDowell, A CASE tool for reusable software component storage and retrieval in rapid prototyping, Information and Software Technology, England, Vol. 38, No. 9, Nov. 1991, pp. 698-706
6. Luqi, V. Berzins, M. Shing and N. Nada, Evolutionary computer aided prototyping system (CAPS), to appear in the Proceedings of the TOOLS USA 2000 Conference, Santa Barbara, CA, July 30 - August 3, 2000

7. Jini technology architectural overview, URL: http://www.sun.com/jini/whitepapers/architecture.html, retrieved 07/31/2000.

8. V. Berzins, Luqi, B. C. Shultes, et al, Interoperability technology assessment for joint C4ISR systems, Technical Reports, Naval Postgraduate School, Monterey, CA, USA, October 1999

9. Luqi and M. Shing, real-time scheduling for Software prototyping, Journal of Systems Integration, Vol. 6, No. 1-2, pp. 41-72, 1996

10. Luqi, V. Berzins, M. Shing, N. Nada and C. Eagle, Computer aided prototyping system (CAPS) for heterogeneous systems development and Integration, to appear in the Proceedings of the 2000 Command and Control Research and Technology Symposium, Naval Postgraduate School, Monterey, CA, 26-28 June 2000

11. Jiang Guo, Luqi, Toward automated retrieval for a software component repository, Proceedings of IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS), Nashville, USA, March 7-12, 1999. Pp. 99-105

A FRAMEWORK FOR NATURAL LANGUAGE AGENTS

John Gregory

Intel Corp.
141 Whiting Way
Folsom, CA 95630
john.p.gregory@intel.com

Du Zhang and Meiliu Lu

Department of Computer Science
California State University
Sacramento, CA 95819-6021
{zhangd, mlu}@ecs.csus.edu

Abstract

Speech technology has been moving ever increasingly into the domain of the everyday computer user. Computer users would use speech technology more readily if they could speak to the machine like they could talk to another person. With advances in visual agent and natural language technologies, this concept is already a possibility. In this paper, we present some ideas about a framework of a type of user interface agent known as a *natural language agent* which combines spoken language understanding and visual agent technologies into a simple to use computer interface. Preliminary results of two experimental agents based on the framework are discussed. Future work on creating complete natural language agent systems is also included.

Keywords: natural language agents, visual agents, speech technologies.

1 Introduction

If you could decide how you wanted to communicate with your computer, would you really pick a keyboard and a mouse as the best way? Instead, what if we could communicate with our computer just like we do with people? We have been trained for many years to use the artifacts of keyboard and mouse to interface with our computers; but that's the whole point, we've been *trained* to use them. Instead, we should be creating computer interfaces that adapt to the way people communicate with each other. In this area, we are on the cusp of a new age in human-computer interaction. The technologies necessary to support human-like communication with a computer are slowly coming of age; and when they do, everyone will be able to easily use a computer.

The next generation of human-computer interaction will allow the user to interact with a computer system using the language they speak to others with every day and they get to choose how the computer will represent itself to them. Getting a system to use spoken language as an interface is just one piece of the puzzle. In order to effectively communicate, most human beings require a visual representation of who or what they are speaking to in order to feel comfortable with this means of communication. Visual agents are a natural fit for this responsibility. By creating a visual avatar for the computer to use as the interface, the user feels more comfortable with the interaction because now they are talking to *somebody*. Additionally, a visual avatar can use such techniques as body language and other body movements to communicate on another level with the user, just like human beings do [1]. Thus, *natural language agents* (NLA) refer to a type of user interface agent that combines spoken language understanding and visual agent technologies to create a simple to use computer interface.

This paper proposes a framework for NLA in the Microsoft (MS) Windows environment and discusses some preliminary results. Section 2 covers the state of the component technologies for NLA as they stand today. Section 3 describes the proposed natural language agent framework called Secret Agent. Some preliminary results based on the Secret Agent framework are given in Section 4. Finally, Section 5 concludes the paper with some remarks on future work.

2 Natural Language Agents

We have been exposed to natural language agents of all types through TV and movies over the years. However, there are many advances in fields other than computer science, which are necessary to support that level of technology. In the meantime, natural language agent computer interfaces can be created using technology available today that will allow an ordinary person to communicate with their computer just like it is another person. Ultimately, the agent could become the user's everyday friend and helper.

The components necessary to create a basic natural language agent include 1) a user-selected visual agent representation, 2) speech recognition, 3) speech synthesis, 4) natural language understanding, 5) an interface to the system the agent is designed to help with, and 6) some additional utility functions. The visual agent gives the user a visual persona to which they can speak with during their interaction in order to increase the comfort of communicating with a computer. By giving the user control over the visual representation of the agent, the

100

user can customize and select the representation that is most entertaining or interesting to work with. Next, the speech recognition component allows the agent to translate the physical speech utterances of the user into meaningful words in the user's language. Also, the speech synthesis allows the agent to speak back to the user for complete spoken interaction. Then, the natural language understanding component is necessary in order to translate the words that a user speaks into ideas and concepts, so that the user can make meaningful requests or have a conversation with the agent. Finally, an interface to the system, which the agent is helping with, allows the agent to enact the requests that the user might make during a session with the agent. Additionally, more components can be added to the agent to increase its functionality and usefulness, including long-term memory, adaptation of conversation to user preferences and work habits, conversational capabilities, and others. For some of the technologies that were just discussed, there are a number of options available for Microsoft Windows-based components, such as:

- Visual Agent – MS Agent [2], CSLU Baldi [3]
- Speech Recognition/Synthesis – MS Speech SDK [4] supporting IBM, Dragon, MS, and Lernout and Hauspie speech engines

For natural language understanding, the field is still in the research phase (see MIT [5] and CMU [6]) though some expensive commercial work is being done today by Cycorp [7]. The remainder of the natural language agent components will need to be custom built until natural language agent technology becomes more common.

3 Secret Agent Framework

In general, an NLA has the structure shown in Figure 1. It includes all the components discussed in Section 2 and interfaces with both the application and operating system. The Secret Agent framework (SAF) is designed to encapsulate the visual agent and speech technologies that are necessary for any NLA application.
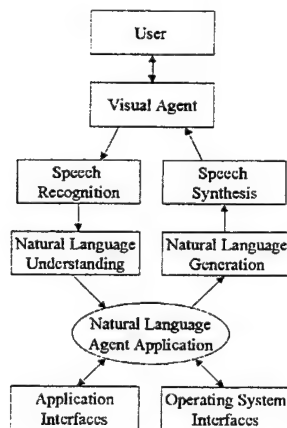

Figure 1. Generic NLA framework.

The goal of the SAF is to base the framework on the most publicly accessible and standardized components that could be found for MS Windows. Since MS Agent and the MS Speech API are the de facto standards for visual agents and speech in MS Windows, they are chosen for the SAF. Figure 2 shows the structure of the modules in the SAF. A separate speech synthesis module is not needed in this case, because it is incorporated into MS Agent. Since the SAF incorporates visual agent and speech technologies, it can be used in any number of applications, such as tutoring and personal assistant applications, that require these technologies.
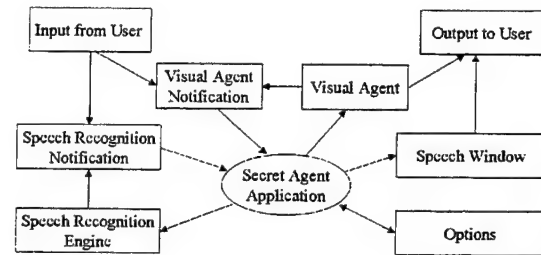
Figure 2. Secret Agent Framework (SAF).

Since both of the visual agent and speech technologies support the MS Component Object Model (COM) [8], the SAF is implemented using C++ and direct COM interfaces for maximum flexibility in control of the COM objects provided by the technologies.

Additionally, the SAF provides a number of user configurable options that are accessible via a dialog built into the framework. Using these options, the user has full control over the agent visual representation, speech recognition engine and speech synthesis voice used for the agent. Also, an optional speech window allows the user to see what the agent has heard so the user knows when the speech engine needs to be trained.

4 Some Example Agents

The first SAF-based NLA is based an old BBS door program called Eliza. Joseph Weizenbaum originally created Eliza as a challenge to the Turing test. Since Eliza is based on the Rogerian mode of therapy in which the therapist strives to eliminate all traces of his of her personality from the dialog, Weizenbaum had planned to show that the test could be beat through the use of 'tricks' instead of true 'intelligence' [9]. A conversation engine is built into the NLA that would mimic the functionality of the original Eliza application. The result is a natural language agent that responds to everything the user says. Depending on the complexity and topic of the conversation, the agent can maintain the illusion of conversationally competence anywhere from 2 responses

to an entire conversation just like the original Eliza. This NLA uses the same algorithms for the conversation engine as the original Eliza application, but adds the extra levels of visual agent and speech technologies.

The Eliza conversation engine is written in C++ as an object, which could communicate with the Secret Agent framework. With the response per utterance mode that Eliza works in, the objects are easily integrated. The only other key feature of interest is that Eliza uses a word matching heuristic to approximate conversational competency. Every word pattern handled by the conversation engine is associated with a standard response that may employ words in the user's original utterance. If words from the user's utterance are used, the words involved are conjugated and transposed to make the response fit the user's input. This is what gives the user the perception of speaking to a psychiatrist. The word patterns and responses are stored in a configuration file that is loaded by the application at startup, and can be easily modified and expanded.
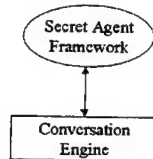

Figure 3. Conversational NLA.

The second SAF-based NLA stems from the fact that users will want a natural language agent to help control the applications that they use everyday. In this respect, we choose a web browser as the application of choice. Two factors are behind the selection of a web browser application: the high demand for web centric applications in today's market, and the availability of a web browser application interface. Using the SAF as the basis, command-understanding capability, an interface to a web browser and some limited OS interaction are added. To approximate natural language commands, the following methods are prototyped: continuous dictation, frame-based grammar and a standard grammar. A standard grammar is created for this NLA due to the simplicity of creation and lack of ambiguity of speech during use. It is also a standard natural language approximation technique used by most modern speech recognition applications.

The technical work on the web browser agent is quite a bit more complicated due to the interface with an independent commercial application. Microsoft Internet Explorer is chosen as the web browser in the experiment, since MS provides classes that encapsulate a programmable interface to the browser using COM technology [10].

Most of the functions in the programmable interface are enabled in this NLA. These functions include: simple navigation commands (back, home, forward, etc.), scrolling capability and application control (toolbars, modes). Expansion is made to the functionality by allowing the user to navigate hyperlinks on a page through spoken commands. There are two parts to this feature. For text links, a routine is called after a page is loaded to dynamically update the grammar used by the speech recognition engine. For other links (such as pictures), another routine intercepts the incoming HTML page and adds numbers to each of the hyperlinks on the page, which can then be spoken to navigate to those links. Additional application capability is added to allow the user to verbally select buttons on dialog boxes that might come up during a typical web browsing session. Finally, a simple help section is added which outlines how the agent works as well as a list of supported commands. All the help and command information is stored in text files which can be modified and expanded. A detailed discussion on these two experiments can be found in [11].
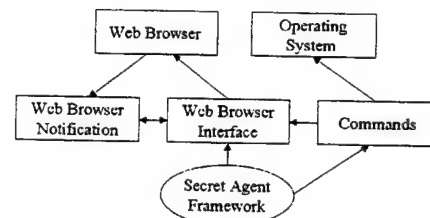
Figure 4. Web browser NLA.

5 Conclusion

The two example agents are just the tip of the iceberg of what can be done with the SAF and other technology available today. The conversation agent could be programmed with a better natural language paradigm to allow it to interact in a more realistic way with the user. The web browser agent could be enhanced by creating interfaces to more applications (e-mail, word processing, etc.). Both of these experiments represent only two facets of the ultimate goal for NLA: to create conversationally competent NLA that can be used as the complete interface to a system.

In order to achieve the goal of NLA, there are a number of things that need to happen. First, natural language understanding engines need to be created which can succeed in the domains that users want to use natural language technology. Next, any application that would like to interface with a natural language agent needs to provide an interface through which the agent can control

102

the application. Finally, the agent needs to be able to make the user feel comfortable by being able to learn and act like another person. Once these steps are achieved and integrated, NLA agents will become common fare.

So what's next? In the near future, NLA or comparable technology will be a standard OS component in the consumer computing marketplace. Already, products such as IBM ViaVoice Millennium [12] are filling the void by creating the first commercial versions of NLA.

References

[1] "Intelligent User Interfaces", about.com (09/28/99), Accessed Nov. 7, 1999, http://ai.about.com/compute/software/ai/library/weekly/aa092899.htm

[2] "Microsoft Agent Home Page", Microsoft Corporation, Accessed Nov. 6, 1999, http://msdn.microsoft.com/msagent

[3] R.A. Cole, "Tools for Research and Education in Speech Science", Center for Spoken Language Understanding, Accessed Nov. 6, 1999, http://cslu.cse.ogi.edu/tm/ron_icphsl.html

[4] "Intelligent Interface Technologies - Speech Application Programming Interface (SAPI)", Microsoft Corporation, Accessed Nov. 6, 1999, http://microsoft.com/iit/projects/sapisdk.htm

[5] "MIT: Spoken Language Systems", Massachusetts Institute of Technology, Accessed Nov. 6, 1999, http://www.sls.lcs.mit.edu/sls

[6] "Language Technologies Institute", Carnegie Mellon University, Accessed Nov. 6, 1999, http://www.lti.cs.cmu.edu

[7] "Cycorp: Makers of the Cyc Knowledge Server for artificial intelligence-based Common Sense", Cycorp Inc., Accessed Nov. 6, 1999, http://www.cyc.com

[8] S. Williams, and C. Kindel, "The Component Object Model: A Technical Overview", Microsoft Corporation, Accessed Nov. 6, 1999, http://msdn.microsoft.com/library/techart/msdn_comppr.htm

[9] J. Weizenbaum, "Eliza", Communications of the ACM, 9:36-45, 1966

[10] "Reusing Browser Technology", Microsoft Corporation, Accessed Nov. 6, 1999, http://msdn.microsoft.com/workshop/browser/default.asp

[11] J. P. Gregory, Natural Language Agents, Master Degree thesis, Department of Computer Science, California State University, Sacramento, December 1999.

[12] "ViaVoice", International Business Machines (IBM) Corporation, Accessed Nov. 6, 1999, http://www-4.ibm.com/software/speech

Implementing Metcast in Scheme*

Oleg Kiselyov

Software Engineering, Naval Postgraduate School, Monterey, CA 93943
oleg@pobox.com, oleg@acm.org

Abstract

This paper presents a case study of implementing a large distributed system in Scheme. *Metcast* is a request-reply and subscription system for dissemination of real-time weather information. The system stores a large amount of weather observation reports, forecasts, gridded data produced by weather models, and satellite imagery. A Metcast server delivers a subset of these data in response to a query formulated in a domain-specific language. Decoders of World Meteorological Organization's data feed, the Metcast server, XML encoders and decoders, auxiliary and monitoring CGI scripts are all written in Scheme.

This paper considers two examples that demonstrate benefits of our choice of the implementation language: parsing of the data feed and a module system for the Metcast server. We will also discuss extensions to Scheme as well as performance.

1 Overview of Metcast

Metcast is a request-reply and a subscription system for distributing, disseminating, publishing and broadcasting of real-time weather information [1]. The system comprises clients and servers communicating in an HTTP protocol. A Metcast server maintains a database of weather observation reports, forecasts, advisories, gridded data produced by weather models, as well as of satellite imagery and plain text messages and discussions. A Metcast client uses a web form or a domain-specific, flexible request language to retrieve a subset of data from a Metcast database [2]. A Metcast server – which is an application (web) server – parses requests, queries the database and sends the requested data in a single- or a multi-part reply. A server may act as a client to request a subset of data for further redistribution. Metcast servers are in operation on several U.S. Navy Meteorology and Oceanography centers worldwide. Clients are deployed on great many sites throughout the U.S. Navy as well as U.S. Air Force, DoD, NATO, NOAA and other government agencies.

One particular source of original data is World Meteorological Organization's (WMO) data feed, containing a great number of land and sea surface and depth/height profile reports, forecasts, advisories, discussions, etc. – for the whole globe. A set of decoders processes the feed, and stores

raw and decoded data in a database. A Metcast server distributes this information in an XML OMF format [3].

The Metcast server, the set of decoders for various WMO data formats, auxiliary and monitoring CGI scripts are all written in Scheme. Metcast clients are written in C++, Java, Scheme, Perl, Python, JavaScript, and Visual Basic.

The server and related modules are implemented in 12800 lines of Scheme code, counting the comments. WMO data feed decoders add 8400 more lines. The size of common extension libraries is 5400 lines of Scheme and some embedded C code. A Gambit-C 3.0 Scheme interpreter enhanced with compiled-in extensions has been used throughout the project.

2 Parsing of the data feed

Scheme proved to be particularly helpful in parsing of the WMO data feed. WMO code is a rather old, ad hoc, peculiar, somewhat inconsistent, tangled data format with a number of options, exceptions and special cases. Furthermore, received bulletins often contain errors due to manual miscoding and transmission problems.

A typical WMO report – for example, a surface synoptic report – is a sequence of code groups separated by white space. A code group is a string of letters, numbers and a few special characters. A code group or groups encode the result of observation of a particular quantity, e.g., cloud conditions, temperature, etc. If code groups were atomic tokens, a report could easily be parsed by a LR(1) automaton. Alas, code groups are composite entities that encode information in idiosyncratic ways. The mere identification of a code group depends on its position and context, which may encompass all previously seen code groups.

We have implemented a report decoder as a combination of a table-driven automaton and code-based group parsers. The latter recognize, parse, and validate a particular code group. The decoder takes a list of code groups and returns an associative list, an Abstract Syntax "Tree" (AST). A special procedure later walks the AST and records the parsed data in a database upload buffer. Of a particular help was Scheme's ability to store and pass procedural values as any other values. This let us implement decoders as *compositions* of code group parsers. For example, a very typical production <a>? <b>* <c>? can be parsed by a combination (sequence parse-a (sequence (loop parse-b) parse-c)). This composition of group parsers is represented by a list (parse-a (repetition-flag parse-b) parse-c). Given this list and the list of code groups to decode, a main driver walks both lists, applying the current parser to the current code

group. The result of the application as well as the repetition flag determine if the current code group is consumed, if the next parser should be chosen, and how AST should be extended.

All the group parsers have the same interface. They receive as arguments the current code group and the AST, and should return:

- an association (a name-value pair) or a list of such associations to add to the AST;

- a symbol pass if the parser failed to recognize the code group. The code group should be given to the next parser;

- #f meaning a syntax error is detected at the current token;

- a symbol terminate to stop parsing of the report.

In the successful case (the first one above), the current token is assumed consumed. Any group parser may examine the AST (that is, the results of the previous parsers) and may even modify the AST. Therefore our parsing technique is somewhat similar to attribute grammars. Figure 1 shows an example of a group parser.

The example demonstrates an and-let* construction (SRFI-2), which was used frequently throughout the project and proved very helpful. As Fig. 1 shows, once the current token has been recognized as a potential <temperature-dew-point> group. and-let* carries on a sequence of elementary parsing decisions, all of which must succeed.

The Metcast decoder is continually processing incoming files, which are delivered every 1-3 minutes. A rather large batch of reports – 8 plain-text bulletins, 144 sea surface observation reports, 777 upper-air level data, 2 terminal airdrome forecasts and 322 synoptic reports – takes 8 wall-clock seconds to parse and 19 seconds to upload and record into the database. The platform is Sun Enterprise-450 server with two UltraSPARC-II CPUs and 512 MB RAM, running Solaris 2.6 and Informix 7.3 database. Keeping in mind that incoming reports have up to 10-minute delay from the time of issue, the total processing time at the Metcast end – under 1 minute – is entirely acceptable.

3 Implementing the Web application server

Scheme turned out to be a good implementation language for a web application server as well. One part of the server is a complex finite state machine that decides when a multipart reply is called for, and sends the corresponding MIME headers. The problem is not trivial as it is generally impossible to predict the number of non-empty replies for a complex request. Expressing such finite automata as sets of mutually-recursive procedures made the code clear and flexible.

Scheme was conducive to compilation and interpretation of the S-expression-based Metcast Request Language [2]. A request language phrase is compiled into a dictionary – an ordered sequence of bindings, – which constitutes the *environment* to look up all data needed to construct a Metcast database query. This hierarchical repository follows neither the static scope of Scheme expressions, nor the dynamic scope of procedure activations. Some bindings may be to procedures, which may push additional associations into the environment and thus affect further lookups.

Metcast server has a highly modular structure. The main program is responsible for receiving and parsing of a request,

and packing of replies. Execution of a particular product request is delegated to a separate module (plug-in). The hierarchical repository was indispensable in implementing a *parameter bus*, which maintains the configuration for the main server and all plug-ins. The parameter bus also provides a uniform interface for invocation of modules and passing of a complex set of explicit and default parameters. For example, the main Metcast server module contains a form (include "metar.scm") that loads a plug-in metar.scm. The latter file defines procedures perform-metar-request and perform-MSL-request. The file binds these procedures to the corresponding Request Language verbs and the configuration information:

```
(env#bind*
 '((METAR (executor . ,perform-metar-request)
       (mime-type . "text/x-omf"))
   (MSL (executor . ,perform-MSL-request)
       (mime-type . "text/x-msl"))
   (OBJ-LOADER:st_constraint .
       ,(lambda constr-l
       (env#bind st_constraint constr-l))) ))
```

When metar.scm is loaded, the above initialization expression is evaluated. The Metcast server thus gains an ability to process requests for METAR and MSL products. The main server module contains a long chain of (include "xxx.scm") expressions, which define a set of requests a server accepts. Adding or replacing support for a particular product requests is as simple as loading or reloading the corresponding plug-in. This re-configuration and linking-in of the modules is possible while the server is running – although we have not pursued this opportunity. The flexible module linking mechanism was beneficial even in the static case as it made incremental development and evolution of the server easier.

4 Extensions to Scheme

Implementing Metcast required several extensions of the Gambit-C Scheme system: libraries of common procedures, and interfaces to external applications and the OS. Detailed descriptions for all extensions along with the commented source and validation code are freely available from a web site [4].

We have already mentioned one helpful extension: and-let*, an AND with local bindings, a guarded LET* special form. An input parsing library was another extension. It is a set of procedures that either skip, or build and return tokens following inclusion or delimiting semantics. The input parsing library has been used on very many occasions: in splitting WMO data feed files into bulletins and bulletins into code groups; in parsing of a QUERY_STRING or HTML form POST submissions; in breaking the response stream from a database query into rows and columns of data; in parsing of XML.

Another kind of extension – made possible by Gambit's excellent Foreign Function Interface – deals with accessing processes, files, directories, communication pipes and other objects external to a Scheme system. Scanning of a POSIX directory is implemented in a truly Scheme style and spirit: The OS:for-each-file-in-directory iterator combines the best features of for-each, map, and filter, and permits premature termination of iterations.

A very helpful extension that goes far beyond Scheme is opening and communicating through uni-, bi-directional, and TCP pipes as if they were regular files. This extension allows Scheme code to talk to external applications or

```
; <temperature-dew-point> ::= <temp> "/" <dew-point>?
; <temp> ::= "M"? <two-digits>     <dew-point> ::= "M"? <two-digits>
(lambda (token AST) ; "/" must be either in the pos 2 or 3
   (let ((slash-pos (string-index token #\/)))
     (if (not (memv slash-pos '(2 3))) 'pass
       (and-let*
         ((negate (lambda (x) (and x (- x))))
          (tempr
            (if (char=? #\M (string-ref token 0))
              (negate (string->integer token 1 3))
              (string->integer token 0 2)))
          (dp-pos (++ slash-pos))
          (dp (if (>= dp-pos (string-length token)) 'none
                (if (char=? #\M (string-ref token dp-pos))
                  (negate (string->integer token (++ dp-pos) (+ 3 dp-pos)))
                  (string->integer token dp-pos (+ 2 dp-pos))))))
         (if (eq? dp 'none)
           (cons 'T tempr)
           (list (cons 'T tempr) (cons 'DP dp)))))))
```

Figure 1: A <temperature-dew-point> group parser

internet services. One particular kind of such an external application is a command-line SQL tool, which allowed us to build a portable database access library [4]. A database query interface is implemented in a Scheme spirit as well, as a general iterator over a collection of selected rows.

5 Illusory and real difficulties

Choosing an implementation language other than C or C++ inevitably raises the question of performance. We have run several benchmarks to ascertain the total performance and its contributing factors. For example, a sample request that retrieves 707 WMO messages (totaling 821K of output) took 25.6 sec (real), 24.1 sec (user) and under 0.1 sec of the system time. This running time comprises: loading and interpretation of the Metcast server script, database connection and query, Request Language interpretation, and output formatting. We conducted several experiments to isolate each of these factors, on the Sun E450 platform described above.

Connecting to a database with a SQL command-line tool dbaccess and running the query took 1.3 sec (real) and 1.0 sec (user). Thus the database interface – however ugly and inefficient it looks – is not the bottleneck. Parsing of the database reply in (interpreted) Scheme code adds 3.8 sec (real) and 2.2 sec (user) time. That is noticeable yet insignificant compared to the total time above. Instrumentation of the Metcast server showed that the server start-up time is under 1.0 sec of real time. This fact was one of the two biggest surprises. Given the complexity of the start-up process – launching of the Gambit interpreter, reading of the main script and 15 included scripts totaling 12800 lines of code. macro-expansion and byte-compilation – one would have expected the start-up to be a significant factor if not the bottleneck. The other biggest surprise was the fact that the most of the running time – 20 seconds – was spent within 7 lines of code, which copy characters from one stream to another while unescaping newlines. A makeshift optimization – copying streams line-by-line rather than character-by-character, and utilizing Gambit's undocumented function ##write-substring – reduced the benchmark real running time from 25.6 sec down to 17.0 sec.

6 Conclusions

Implementation of a web application server and WMO decoders in Scheme showed that the language is up to the task. The elegance of Scheme and its ability to easily express guarded execution, finite-state machines as sets of mutually recursive actions, hierarchical repositories with procedural bindings turned out to be most important. Built-in garbage collection, iterators, safety, the ease of incremental testing cannot be overestimated either. Despite obvious inefficiencies, so far overall Metcast server performance is deemed satisfactory by customers.

References

[1] Oleg Kiselyov, "Distributing Weather Products through an HTTP pipe" http://zowie.metnet.navy.mil/~spawar/JMV-TNG/ http://pobox.com/~oleg/JMV-TNG/ March 10, 2000.

[2] Oleg Kiselyov, "A delegation language to request weather products and a scheme of its interpretation," Proc. third ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'98), Baltimore, Maryland, Sep. 27-29, 1998, p. 343.

[3] Oleg Kiselyov, "Weather Observation Definition Format" http://zowie.metnet.navy.mil/~spawar/JMV-TNG/XML/OMF.html March 8, 2000.

[4] Oleg Kiselyov, "Scheme Hash," An archive of Scheme code http://pobox.com/~oleg/ftp/Scheme/ July 4, 2000.

Towards an Ontology of Software Maintenance

BARBARA A. KITCHENHAM[1]*. GUILHERME H. TRAVASSOS[2], ANNELIESE VON MAYRHAUSER[3],
FRANK NIESSINK[4], NORMAN F. SCHNEIDEWIND[5]. JANICE SINGER[6], SHINGO TAKADA[7].
RISTO VEHVILAINEN[8] and HONGJI YANG[9]

[1] *Department of Computer Science. Keele University. Staffordshire. ST5 5BG, U.K.*

[2] *COPPE/UFRJ, BR and DCS/ESEG. University of Maryland, A. V. Williams Bldg.. College Park MD 20742, U.S.A.*

[3] *Computer Science Department. Colorado State University. 601 S. Howes Lane, Fort Collins CO 80523-1873, U.S.A.*

[4] *Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1018 A. 1081 HV Amsterdam, The Netherlands*

[5] *Naval Postgraduate School. 2822 Racoon Trail, Pebble Beach CA 93953, U.S.A.*

[6] *Institute for Information Technology. National Research Council. Ottawa ON K1A 0R6, Canada*

[7] *Dept of Information and Computer Science, Faculty of Science and Technology, Keio University, 3-14-1 Hiyoshi. Kohoku-ku, Yokohama. Kanagawa 223-8522, Japan*

[8] *DPM Consulting Oy. Haukantie 21 A, 04320 Tuusula, Finland*

[9] *Computer Science Department. De Montfort University. Leicester. LE1 9BH, U.K.*

SUMMARY

We suggest that empirical studies of maintenance are difficult to understand unless the context of the study is fully defined. We developed a preliminary ontology to identify a number of factors that influence maintenance. The purpose of the ontology is to identify factors that would affect the results of empirical studies. We present the ontology in the form of a UML model. Using the maintenance factors included in the ontology, we define two common maintenance scenarios and consider the industrial issues associated with them. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: empirical research; maintenance factors; maintenance scenarios; evolutionary maintenance: independent maintenance groups; maintenance ontology

1. INTRODUCTION

This paper arose from a discussion session held at the 3rd Annual Workshop on Empirical Studies of Software Maintenance ('WESS '98'). The task of the session was to consider the question 'What are the differences between maintenance tools/methods/skills and those of development?' From the point at which members of the group stated their preliminary positions, it was evident that we would find it difficult to give a single answer. The position statements ranged from what can be paraphrased as 'Nothing much' to 'Lots of stuff.'

*Correspondence to: Dr. Barbara A. Kitchenham. Department of Computer Science, Keele University, Staffordshire ST5 5BG, U.K. Email: barbara@cs.keele.ac.uk

As the discussion continued. it became clear that our difficulties arose from our different views of what constituted 'maintenance'. We concluded that we could not answer any serious questions about maintenance methods, tools or skills until we had a description of maintenance rich enough to encompass all our different experiences of maintenance. We concluded that what we needed was an ontology of maintenance—that is. a specification of a conceptualisation (Gruber, 1995). This ontology should not be only a hierarchy of terms. but a framework talking about the maintenance domain and identifying the factors that affect maintenance. supported by a taxonomy describing the different factor levels.

We believe that such an ontology would have four major benefits for the maintenance research community. It would:

1. allow researchers to provide a context within which specific questions about maintenance can be investigated:
2. help to understand and resolve contradictory results observed in empirical studies;
3. provide a standard framework to assist the reporting of empirical studies in a manner such that they can be classified. understood and replicated: and
4. provide a framework for categorising empirical studies and organising them into a body of knowledge.

Furthermore. if we could report our research results in a systematic fashion. clarifying the context to which the results apply. it would also help industrial adoption of research results.

In Section 2. we present an overview of the ontology. In Section 3 we describe our proposed maintenance ontology in more detail. In Section 4. we look at two maintenance scenarios and consider how the ontology can be used to help characterise the difference between the scenarios.

2. OVERVIEW

de Almeida. de Menezes and da Rocha (1998) describe the process of constructing an ontology as involving the following activities:

- purpose identification and requirement specification:
- ontology capture and formalisation:
- integration of existing ontologies: and
- ontology evaluation and documentation.

Knowledge captured in an ontology is usually represented in a graphical notation. For instance. GLEO (Graphical Language for Expressing Ontologies) was used to describe a software process ontology (de Almeida. de Menezes and da Rocha. 1998).

In this paper, we consider only a part of the ontology construction process. We consider only purpose identification and requirement specification and ontology capture. Moreover. since we do not intend to provide a formal description. we present our ontology in a subset of UML (Unified Modelling Language) notation (Fowler and Scott. 1997) instead of GLEO. UML has been used by other researchers to describe knowledge. For example, Hasselbring (1999) used UML to describe knowledge concerned with health care information systems. Since UML is a standard object-oriented notation. we believe it will make our ideas more accessible to software engineering and
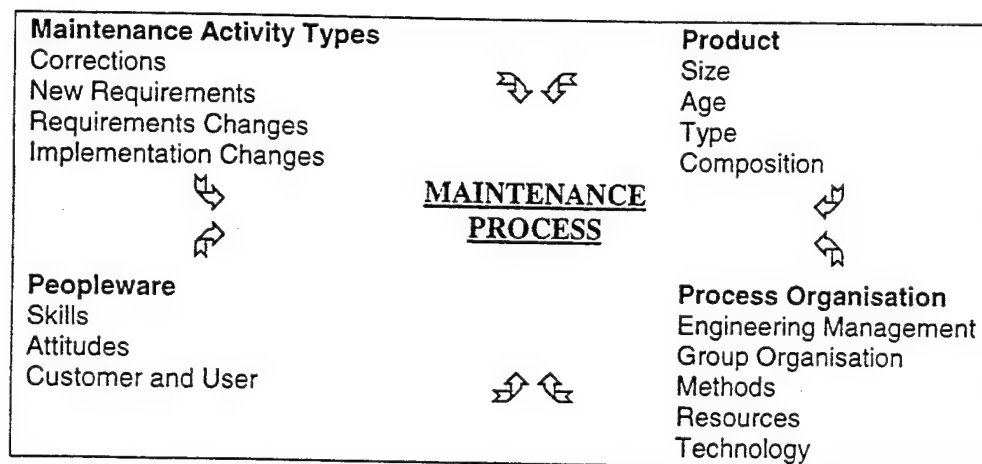
108

Figure 1. Overview of domain factors affecting software maintenance

software maintenance researchers. Furthermore, it is possible to improve the representation of the ontology at a later date by inserting the axioms needed to formalise the whole model.

As a result of our discussions at the WESS 98 workshop, we identified a number of domain factors that we believe influence the maintenance process. Figure 1 shows these factors and how they can be classified. Figure 1 was the starting point for our ontology, which is described in more detail in Section 3. In order to describe empirical maintenance research, we believe that the maintenance factors must be specified. This will allow researchers to better understand the maintenance context and to plan the research needed to investigate the relationships among these factors and the maintenance context. A better understanding of the relationships that exist between factors and context should lead both to improvements in the maintenance process and to the development of new research topics.

The maintenance process describes how to organise maintenance activities. It is similar to the software development process, but the focus is on product correction and adaptation, not just on the transformation of requirements to software functionality. We take the same viewpoint when considering methods and tools. It is not usually necessary to define new methods or tools to accomplish maintenance activities; conventional software development tools are usually sufficient. However, the maintenance process defines how these methods and tools should be applied to maintenance activities, and which skills and roles are necessary to carry out the activities. Previous research work has considered the definition of methods (Karam and Casselman, 1993), process description (Pfleeger, 1998), software environment ontology (de Almeida, de Menezes and da Rocha, 1998), and tool classification (Pressman, 1997). Although these research results considered the software development process as the basic framework, they are also useful in the context of the maintenance process.

In order to understand the relationships among maintenance domain factors, we need to specify each factor and define the impact that it has on maintenance activities. Next, the relationships themselves can be captured and validated. Validation usually requires empirical studies and experiments.

Figure 1 has some similarities with the framework for software maintenance suggested by Haworth, Sharpe and Hale (1992). They defined a framework based on four entities: programmer, source code, maintenance requirement and environment. They suggested that each of these basic entities in the framework interacted to a degree with the other entities. Each of the entities and each combination of possible interactions contribute to a research area and define the type of attributes that can be manipulated. For example, one area of research is source code attributes, and another is the interaction between source code attributes and programmer attributes. They use the areas to classify existing research and discuss the way in which experiments aimed at considering interactions could be designed. In our ontology, we have generalised the concepts of maintenance requirement, source code and programmer to maintenance activity, product, and maintenance engineer respectively. We have also introduced another concept: the maintenance organisation process. We have omitted an environment entity because our more generalised concepts include environmental considerations. The main difference between the Haworth, Sharpe and Hale framework and our ontology is that they are concerned with the structure of empirical experiments. So, they are not concerned with the nature of the attributes attached to each of their entities, whereas our main concern is the attributes and the way in which they define the context of empirical research.

3. THE MAINTENANCE ONTOLOGY

3.1. Purpose specification and requirements specification

Before discussing our conceptualisation of the maintenance domain, we need to consider the first stage of ontology development, which is purpose specification and requirements specification. de Almeida, de Menezes and da Rocha (1998) define the activity of purpose specification to be 'to clearly define its purpose and intended uses, that is, the competence of the ontology'. The competency of the ontology identifies the questions the ontology is meant to answer.

In our case, the purpose of our ontology is to identify contextual factors that influence the results of empirical studies of maintenance. For example, suppose a researcher were investigating the impact on productivity of new maintenance tools but did not specify the experience of the tool users. In this case, it would be difficult for other researchers to replicate the study, or for practitioners to know whether or not the results were likely to apply in their own situation. Furthermore, it is not just the experience of tool users that is likely to affect the study's results and their interpretation. Other factors that need to be specified include the type of product being maintained, and the type of maintenance tasks being performed.

In observational studies of maintenance, researchers measure maintenance performance characteristics such as the quality of maintained products, or the productivity or efficiency of the maintenance process for different products or different maintenance activities, in order to identify how and why these performance characteristics vary. In controlled experiments, researchers investigate the impact of one or more factors that they believe affect maintenance quality or productivity by varying the factors in a systematic fashion, while controlling other factors.

Thus, in order to support empirical studies of both kinds, each factor in our ontology needs to answer the following competency question:

Would variations in this factor (i.e., concept) influence empirical studies of maintenance productivity, quality or efficiency?

For the purposes of ontology capture, we do not believe it is necessary to identify every possible interaction between maintenance factors and maintenance performance. However, we do need to present a reasoned argument explaining at least one interaction for each factor. This can also be regarded as a contribution to ontology evaluation. Any such explanation would depend on being able to identify the way in which each element can vary in different circumstances. This implies a second competency question:

What is the nature of the variations in this factor?

This second question leads to preliminary taxonomies of maintenance elements. The taxonomy is also intended to help practitioners identify whether or not empirical results are likely to be relevant to their specific maintenance situation. The two competency questions already identified are sufficient to represent the viewpoint of practitioners as well as researchers.

Finally, we hoped that our taxonomy would also cast some light on our original workshop goal, which was to consider the differences between maintenance and development from the viewpoint of skill, tools and methods. This leads to a third and final competency question:

To what extent do maintenance methods/tools/skills differ from those of development?

To address this question fully, we would need a software process ontology as well as a maintenance ontology. Thus, we have not addressed this competency question fully. We do, however, point out some of the differences we found between our maintenance ontology and the de Almeida, de Menezes and da Rocha software process ontology, and identify some concepts that are of relevance only to maintenance.

The following sections define our ontology. Because the domain is very complex, we describe each main dimension shown in Figure 1 separately, with the final integrated ontology shown later in Figure 7. In the next sections we present our ontology of software maintenance with definitions of all the main concepts (i.e., maintenance factors). Where possible, we make use of definitions and concepts used by de Almeida, de Menezes and da Rocha (1998) in their software process ontology. We also consider the different properties of the maintenance factors that impact the maintenance process and can thus affect the results of empirical studies.

3.2. Maintained product

3.2.1. Overview

Figure 2 shows our product ontology. Table 1 defines the concepts used in the ontology. Characteristics of these elements that affect maintenance performance are discussed in the following sections. Note that in their software process ontology, de Almeida, de Menezes and da Rocha do not consider the relationship between the total product and its composite artefacts.

3.2.2. Product size

The size of the product affects the number and organisation of the staff needed to maintain it. Table 2 suggests a coarse-grain size measure for classification purposes. There are relationships
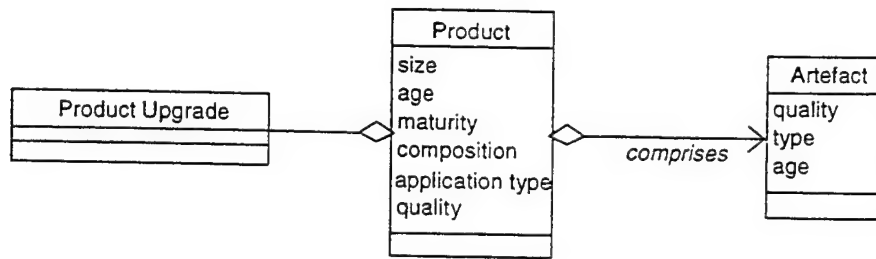
Figure 2. The maintained product ontology

Table 1. The maintained product ontology definitions

Product	The product is the software application, product or package that is undergoing modification. A product is a conglomerate of a number of different artefacts.
Product upgrade	A change to the baseline product that implements or documents a maintenance activity. An upgrade may be a new version of the product, an object code patch, or a restriction notice.
Artefact	Artefacts that together correspond to a software product can be of the following types: documents that can be subdivided into textual and graphical documents, COTS products, and object code components. Textual documents include source code listings, plans, design and requirements specifications.

Table 2. Product size

Product size	Maintenance team size
Small	1 person
Medium	1 team
Large	Multiple teams

between the size measure and maintenance team organisation. For example, geographically distributed maintenance teams usually maintain large products. The size of the enhancements and the size of the product are likely to affect maintenance productivity. The larger the product the more likely it is that product knowledge will be spread unevenly among the maintenance staff, making it more difficult to diagnose the cause of some problems and identify all the modifications needed to support a large enhancement. In addition, when many people are working together on a large enhancement, there are more opportunities for misunderstandings that can lead to quality problems. Thus, maintenance activities on large products may be less productive than maintenance activities on small products.

3.2.3. Application domain

Many researchers (e.g., Maxwell, van Wassenhove and Dutta, 1966) have observed major

J. Softw. Maint: Res. Pract. 11, 365–389 (1999)

112

productivity differences between products from different application domains. We believe such differences apply to maintenance activities as well as development activities. In addition, the application domain (e.g., finance, telecommunications, command and control, etc.) places domain knowledge requirements on maintenance human resources. It also places constraints on the maintenance artefacts and product. For example, safety critical system maintenance must, at all cost, preserve software reliability requirements, whereas in the telecommunications world there is more emphasis on fast upgrades to software in order to minimise time to market. These different constraints mean that different aspects of maintenance performance are optimised.

3.2.4. Product age

The age of a product (i.e., the age in years since first release) can affect maintenance in different ways:

- If the development technology is very old, it may be difficult to find maintenance human resources with skills in the old technology (hence, the practice of 'grey-sourcing' the maintenance of some products by bringing older programmers out of retirement). In addition, it may be difficult to find support tools, such as compilers and static analysers, and support for the tools.
- If the product is old, it may be difficult to access the original developers or the original development documentation. This can lead to products or parts of products that no one understands well enough to change.

Thus, in general we expect maintenance performance to be better for younger than older products.

3.2.5. Product maturity

Product maturity is different from product age. It concerns the life cycle of a product after initial release. The basic phases in the life of a product and their relationship with maintenance tasks and user population are summarised in Table 3, which is similar to the life cycle described by Kung and Hsu (1998). The maintenance life cycle starts at first release and ends when a product is withdrawn from use. It is important to note that large enhancements cause mini-cycles, where a product can be forced back into periods of infancy and adolescence as a result of poor quality product releases. Table 3 suggests that the type of maintenance tasks undertaken by an organisation is related to the maturity of a product, as is the size of its user population. Note that a consideration of user population is irrelevant for some custom-built products that have a single client-single mission profile.

3.2.6. Product composition

The level of abstraction of the component artefacts of a product affects the skills required by maintenance engineers and the tools they need to support them. If products are generated from designs, maintenance engineers need access to the code generation tools. If the product is composed of black box components (e.g., a COTS product), maintenance engineers need integration skills rather than coding skills.

Table 3. Maintenance life cycle

Life cycle stage	Maintenance task prevalence	User population
Infancy—after release, initial users start reporting defects.	Corrections	Small
Adolescence—as the user population grows, defect reports still predominate but there may be changes to amend the system behaviour.	Corrections, requirement changes	Growing
Adulthood—the product is relatively defect free, but if it is accepted by a wide user population there will be requests for new functionality. In addition, as change accumulates there will be a need to restructure parts of the system to avoid design decay, so implementations changes to improve code structure may be required.	New requirements, implementation changes	Maximum
Senility (legacy)—there are newer products available and only a few users remain to be supported. Usually only corrective maintenance and workarounds are provided.	Corrections	Declining

3.2.7. Product and artefact quality

The original software development process and the quality of the product it delivered place constraints on the subsequent maintenance process. In our experience it is easier to maintain a good quality product than a poor quality product, where 'quality' includes issues such as product structure, documentation, and the quality of individual artefacts. Furthermore, the less contact a maintenance organisation has with the original software developers, the more it is dependent on the availability of good quality documentation, bearing in mind that there are many different forms of documentation associated with a software product. In terms of defining the impact of document quality on maintenance activities, we need to assess the extent to which documentation is:

- complete,
- accurate, and
- readable.

For old products, documentation is often poor or non-existent. In such cases, maintenance engineers need specialised tools such as re-engineering tools. Thus, comparisons of maintenance performance across different products will be of limited value unless it is clear that the maintenance tool requirements of each product have been met to an equivalent degree, and that the quality of the component artefacts is comparable.

3.3. Maintenance activities

Figure 3 shows our maintenance activity ontology, which is derived from de Almeida, de Menezes and da Rocha's software development activity ontology. We have amended that ontology to consider maintenance activities rather than software construction activities, and have omitted elements that
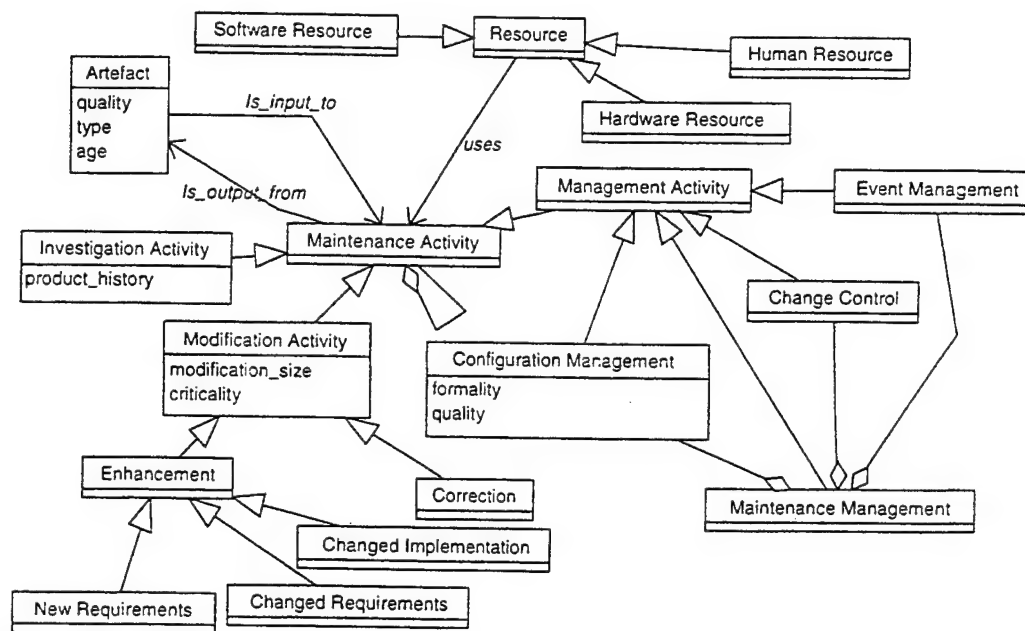
Figure 3. The maintenance activity ontology

do not have any major impact on maintenance performance. In particular, we have added the concept of an investigation activity, and, instead of having a construction activity, we have a maintenance activity. Furthermore, we have identified configuration management as one of the types of management activity. We have also included the resource concept in this ontology, whereas de Almeida, de Menezes and da Rocha (1998) had a separate resource ontology. Definitions of the elements in the ontology are given in Table 4. A discussion of the impact of the elements on maintenance performance follows.

In our view, one of the major differences between software development and software maintenance is that development is requirement-driven and maintenance is event-driven. This means that the stimuli (i.e., the inputs) that initiate a maintenance activity are unscheduled (random) events.

Input events usually originate from the users (or client or customer) of the software application, but may also originate from maintenance human resource (engineers or managers). Thus, the first activity needed by a maintenance process (after the administrative process of logging the event) is an investigation activity, whereby a maintenance engineer is assigned to assess the nature of event, which can be either a problem report or change request. On completion of an investigation activity, maintenance managers must decide whether or not to proceed with a maintenance modification. This is discussed in more detail in Section 3.4.3.

Maintenance modifications are often referred to as corrective, adaptive or perfective following Swanson's typology (Swanson and Chapin, 1995). However, since identifying a modification as an adaptive or a perfective maintenance activity depends on the reason for the change, and not on an objective characteristic of the change, we have used the following definition for types of

Table 4. Maintenance activity ontology definitions

Activity	An action of one of the following types: an investigation activity, a modification activity, a management activity, or a quality assurance activity. An activity may be made up of a number of sub-activities. Usually, it takes as input one, or more existing artifacts and outputs zero, one or many new or modified artifacts.
Investigation activity	An activity that assesses the impact of undertaking a modification arising from a change request or problem report.
Modification activity	An activity that takes one or more input artefacts and produces one or more output artefacts that, when incorporated into an existing system, change its behaviour or implementation.
Management activity	An activity related to the management of the maintenance process or to the configuration control of the maintained product (see Figure 5 and Table 6).
Quality assurance activity	An activity aimed at ensuring that a modification activity does not damage the integrity of the product being maintained. Quality assurance activities may be classified as testing or certification activities (entity omitted from Figures 3 and 7).
Resource	Everything that is used to perform an activity. Resources may be hardware, software or human resources.

maintenance changes:

- *Corrections* that correct a defect—i.e., a discrepancy between the required behaviour of a product/application and the observed behaviour.
- *Enhancements* that implement a change to the system that changes the behaviour or implementation of the system. We subdivide enhancements into three types:

    - enhancements that change existing requirements,
    - enhancements that add new system requirements, and
    - enhancements that change the implementation but not the requirements.

Broadly speaking, enhancements that are necessary to change existing requirements can be equated to Swanson's perfective maintenance changes. Those that are necessary to add new requirements to a system can be equated to adaptive maintenance. Changes that do not affect requirements but only affect implementation might be referred to as preventive maintenance (by analogy to what happens when you have your car serviced). Note that corrections may result in similar types of product modifications, but we do not feel that it is necessary to define correction subtypes.

There is not a one-to-one relationship between problem reports and corrective maintenance. Sometimes, the 'problems' noted by users are requests for behaviours that were not originally required. In such cases, the problem report leads to an enhancement rather than a correction. It is important to determine whether maintenance work is a correction or an enhancement because the activities are often budgeted separately. In fact, many of the disputes between the customer/client and maintainers revolve around whether a change is a correction or an enhancement. If the customer/client did not fully and unambiguously define the required behaviour, it is often difficult to decide whether a modification is a correction or an enhancement.

Characteristics of maintenance activities that affect the productivity and efficiency of maintenance activities include the size of the modification and the criticality of the modification. Large enhancements, particularly large enhancements of large products, are likely to require effort from several different maintenance engineers, and will thus incur coordination and communication overheads. Smaller enhancements that can be performed within schedule by one maintenance engineer are usually more productive. The criticality of an enhancement or correction impacts the elapsed time it takes for the modification to be delivered to users, since the scheduling of the modification will be determined mainly by its criticality.

To accomplish the different maintenance activities, maintenance engineers require different degrees of product understanding and different types of development tools. A corrective activity may require only the ability to locate faulty code and make localised changes, whereas an enhancement activity may require a broad understanding of a large part of the product (Singer, 1998). In the first case, a maintainer will require testing or simulation tools to recreate the problem and debugging tools to step through suspect code. In the second case, a maintainer's tool requirements will depend on the quality of the development documentation, and the availability of the development environment. If the maintainer has poor documentation and little of the original development environment, he/she may require re-engineering tools and/or code navigation and cross-referencing tools.

The efficiency and quality of investigation activities depends on the maintenance engineer knowing the current status of patches and planned modifications that apply to the part of the product involved with the new problem report or change request. The availability of such information depends on the effectiveness of the product configuration control and change control process. A good configuration control process is necessary to identify the status of each product component, including information such as the currently applied patches. A formal change control process might slow down the rate at which the maintenance process responds to input stimuli, but may improve the ability of the change control and maintenance processes to preserve the integrity of the product under maintenance and its constituent artefacts.

3.4. Software maintenance process

3.4.1. Two processes

Within a software maintenance department, there are two different maintenance processes:

- the maintenance process used by individual maintenance engineers to implement a specific modification request, and
- the organisation level process that manages the stream of maintenance requests from customers/clients, users and maintenance engineers.

We consider both types of process separately. In order to use terminology similar to that used by de Almeida, de Menezes and da Rocha (1998), we refer to our definition of the first process as the software maintenance procedure ontology (see Figure 4). de Almeida has no equivalent to the second process in his ontology. We refer to the second process as the maintenance organisation process (see Figure 5).
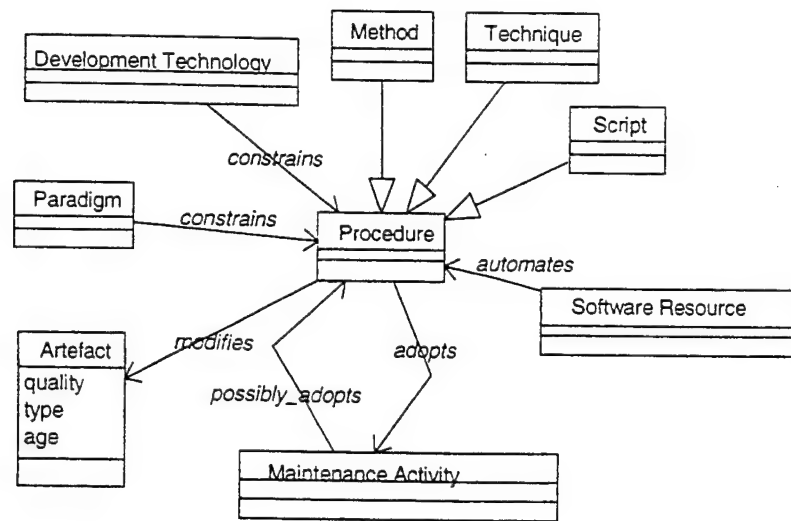
Figure 4. The maintenance procedure ontology

Table 5. Maintenance procedure ontology definitions

Development technology	The technology used when the product and its constituent artefacts were originally constructed. for example. knowledge-based system technology. conventional data processing technology. The original development technology constrains the possible maintenance procedures.
Paradigm	The philosophy adopted during the original construction of the maintained product. for example. the object-oriented paradigm or procedural paradigm. The original paradigm constrains the possible maintenance procedures.
Procedure	The conduct followed to perform an activity. A procedure may be classified as a method. technique or script. A procedure may be adopted to perform a specific activity from a set of possible procedures.
Method	A systematic procedure defining steps and heuristics to permit the accomplishment of one or more activities.
Script	A guideline for constructing/amending a specific type of document.
Technique	A procedure used to accomplish an activity that is less rigorously defined than a method.

3.4.2. *Software maintenance procedure*

The software maintenance procedure ontology shown in Figure 4 is used to modify one or more artefacts in order to implement a required software modification. The concepts shown in Figure 4 are defined in Table 5. The definitions have been adapted from de Almeida. de Menezes and da Rocha's definitions.

Artefacts are not solely source and object code items. They comprise documents, system representations and plans. etc.. constructed throughout the software development process, and

118

modified during maintenance. A variety of different scripts, methods and techniques are used to construct and modify such artefacts, and they are usually available to support maintenance activities.

Maintenance activity performance will be affected by the choice of software development technology and development paradigm. It will also be affected by the extent to which procedures are automated. In general, development technologies such as the development language and the development paradigm place constraints on maintenance activities, and skill requirements on maintenance human resources. The ISO/IEC 12207 Standard defines an 'activity' as a life cycle phase and a 'task' as something done as part of an activity. Here we are using only the term 'activity', but an activity can be decomposed into smaller activities, therefore capturing the ISO/IEC definitions.

In addition, the chosen development technology may present a significant risk to product maintainability. A software product cannot continue to be maintained if its development environment is not available to its maintainers. For products with a long lifetime it is necessary to ensure that technologies such as compilers, code generators and CASE tools will themselves be supported throughout the estimated lifetime of the product.

3.4.3. Maintenance organisation processes

Figure 5 shows the maintenance organisation process. Table 6 briefly defines the concepts used in the model.

A maintenance organisation must handle a stream of maintenance requests from users, customer and maintainers. Thus, a major element of a maintenance organisation is event management (Niessink and van Vliet, 1998). Another major element of a maintenance organisation is configuration management. Configuration management is the process responsible for releasing new system versions and system amendments to users. In addition, configuration control systems need to protect the integrity of the product when it is being modified. In particular, they need to ensure that maintenance engineers know the current repair status of the product and product components. If the configuration control system is inadequate, maintenance activities will be less efficient and there is a danger that product quality will be compromised.

In addition, there needs to be a management process for authorising or rejecting modification activities after initial investigation of the trigger event. This is usually the responsibility of a change control board. The authorisation process may also include a process of negotiation with the client about contractual arrangements for implementing a required modification (e.g., budgets/price and time-scales). Only after a proposed modification activity is approved by the change control board and any necessary contractual arrangements are agreed with the client (which, for applications like operating systems or self-standing products, may be the marketing department), will the proposed modification activity be scheduled. A change control board can be organised as a formal process involving meetings between users and customers/clients and maintenance managers, or as a simple working procedure. The level of formality can affect quality and efficiency. Formal change control boards are likely to slow the maintenance process but are better able to protect the integrity of the product being maintained.

The efficiency of maintenance management activities is affected by the use of support tools. Most organisations have configuration control tools. There are also many tools to assist event management. For example, many maintenance organisations use 'help' desk tools, which allow
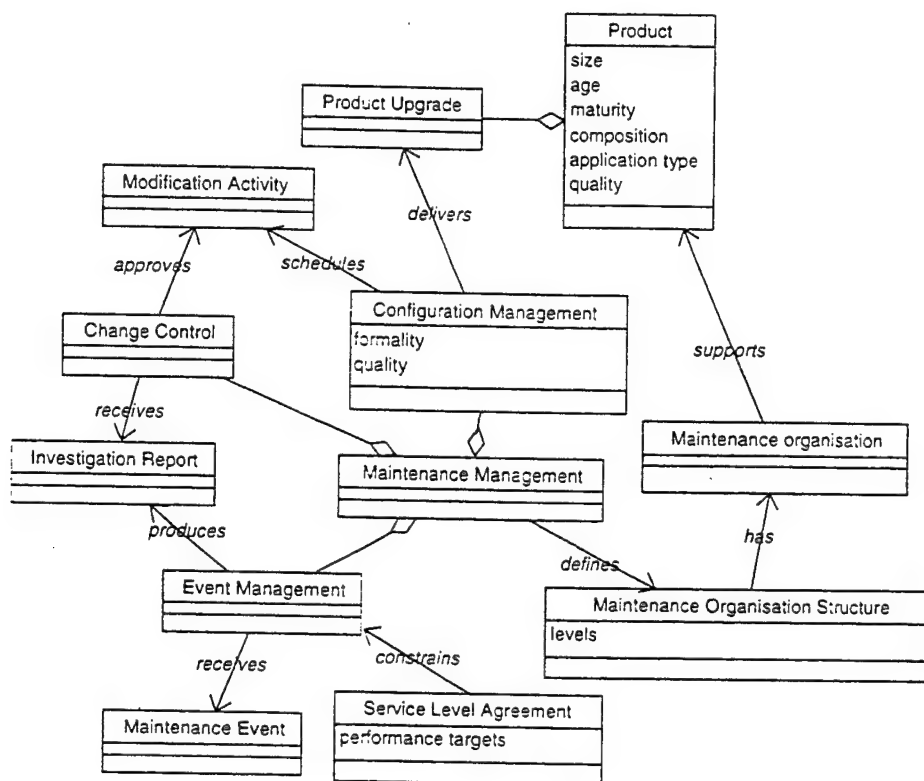
Figure 5. The maintenance organisation process ontology

events to be logged into an organisation and their progress tracked though the various maintenance tasks needed to resolve the event. Another type of tool that supports the interface between the user population and a maintenance organisation is a 'known error log', which identifies all currently known errors and their workarounds or fixes.

The volume and type of maintenance requests affect the performance of the maintenance organisation. For example, if there are a large number of defects reported, there may be insufficient resources to undertake perfective or preventive modifications.

Service level agreements define the maintenance organisation's performance targets. Differences in achieved performance level may, therefore, be due to different performance targets. Maintenance organisations must be engineered to meet their service level agreements. This is often done by separating various support activities into well-defined roles that can be performed by staff with specialised skills. For example, many maintenance organisations use the concept of support levels to separate staff, whose main concern is to support the user population and those concerned with correcting or enhancing software.

At its simplest there may just be two support levels:

- Level 1—this level provides the personnel who staff the help desk.
- Level 2—this level provides the personnel who make changes to software.

Table 6. Maintenance organisation process ontology definitions

Service level agreement	An agreement between the providers of a maintenance service and the customers of a maintenance service that specifies the performance targets for the maintenance service.
Maintenance management	The process used to manage the maintenance service (as opposed to the procedure used to manage individual maintenance requests). The organisation process is established and maintained by senior maintenance managers. It is responsible for defining the structure of the maintenance organisation such that it can fulfill its service level agreement. Maintenance management has three main concerns other than the normal concerns of quality assurance and project management: event management, configuration control, change control.
Event management	Event management is the process responsible for handling the stream of events received by the maintenance organisation.
Change control	Change control is the process responsible for evaluating the results of maintenance event investigations and deciding whether or not to approve a product modification.
Configuration management	Configuration management is responsible for maintaining the integrity of the product in terms of its version and modification status. It is also responsible for the production of product upgrades.
Maintenance organisation structure	The roles undertaken by maintenance human resources in a maintenance organisation in order to perform the required administrative procedures.
Maintenance event	A problem report, or change request originating from a customer or user of the maintained product or a member of the maintenance organisation.
Investigation report	The outcome of investigating the cause and implications of a maintenance event.

However, at least three support levels is the more common situation:

- Level 1—the help desk staff are non-technical, and are responsible for logging problems and identifying the technical support person most likely to be able to assist a user.
- Level 2—the technical support personnel know how to communicate with users and understand their problems, and they can advise on workarounds and quick fixes.
- Level 3—the maintenance engineers are authorised to make changes to the product.

The separation of maintenance services across different service levels makes it clear that not all maintenance work results in product modification. Users may simply require advice about how to use the product or how to circumvent a known problem with the product. The number of levels and the specific roles they support affect the performance of the maintenance service. For example, if there are too many levels there may be an unacceptable delay in responding to certain types of maintenance request.

The other main role for a maintenance organisation is the planning and scheduling of maintenance releases. This involves identifying the content of difference releases and a release cycle that is appropriate to customer requirements. Factors such as the interval between scheduled maintenance releases and the extent of change permitted to a product can have a significant impact on the quality
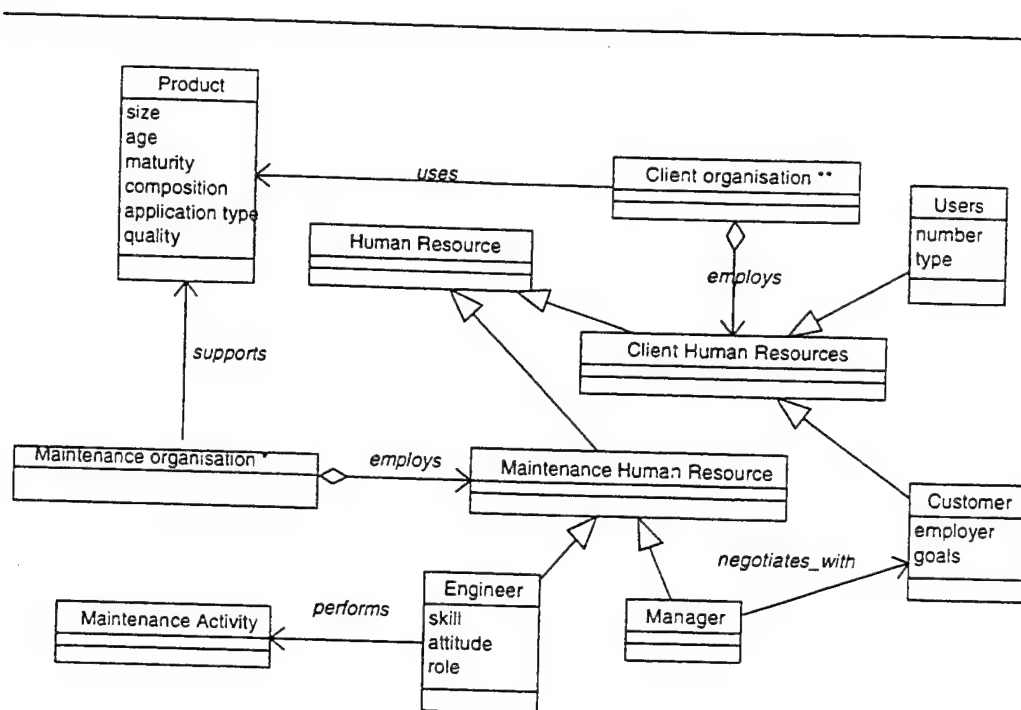
Figure 6. The peopleware ontology

of the maintained product (Lehman, Perry and Ramil, 1998). The procedures for releasing object code fixes (for example, fix on fail, or periodic collated updates) can also affect product quality (Mellor, 1983).

3.5. Peopleware

3.5.1. Two groups

Software production and maintenance are human intensive activities. Furthermore, they involve people working together in teams, which are in turn part of larger organisations. Thus, no complete description of factors affecting maintenance can ignore the human and social elements. There are two types of staff involved in a maintenance process: the staff in the maintenance organisation, and the staff in the customer/client organisation. Figure 6 shows our initial model of these factors. The definition of peopleware concepts is given in Table 7.

3.5.2. Maintenance organisation staff

3.5.2.1. Staff attitudes. Staff attitudes and motivation are generally agreed to impact on the quality of any activity. In the area of software maintenance, problems with motivation are expected because software maintenance is often perceived to be of less importance and less well-rewarded than development.

Table 7. Peopleware ontology definitions

Client organisation	The organisation or organisations that use the maintained product and have a defined relationship with the maintenance organisation.
Maintenance organisation	The organisation that maintains the product or products.
Human resource	Employees of the maintenance or client organisation. Maintenance organisation staff can be classified as managers or engineers. (For simplicity we have omitted specialised QA staff who may be considered a special class of engineer.) Employees of the client organisation can be classified as users or customers. Managers in the maintenance organisation negotiate with customers to determine service level agreements and costs and scheduling of requirement enhancements.

Management often compounds attitude problems by:

- making maintenance work equivalent to a punishment, and
- assigning novices to maintenance work.

This factor seems difficult to characterise. but is likely to have a major impact on the productivity and quality of maintenance activities and the extent to which the maintenance staff is receptive to process change.

3.5.2.2. Staff responsibilities. One area that seems to have a major impact on the entire maintenance culture of an organisation is whether or not there is a strict separation between staff responsible for software development and those responsible for software maintenance.

At one extreme. there is no real separation between development and maintenance. This seems to be associated with a particular type of product. i.e.. a product undergoing continual evolution that is released periodically to clients and users. The software developers incorporate corrective, perfective and preventive maintenance tasks into a process aimed at a continuing stream of planned enhancements. In such an environment there may be no practical difference between the tools and procedures used for 'development' and those used for 'maintenance'. Furthermore. the personnel themselves do not make any significant distinction between development and maintenance. which reduces motivation problems.

At the other extreme, there are maintenance organisations that are completely separate from development departments. and indeed may not work for the same company that developed the code they maintain. In such an environment. maintenance programmers may need specially designed tools to support their maintenance tasks.

Another issue is whether staff are responsible for the maintenance of a single product or group of products (i.e., a product portfolio). It is usual for an evolutionary style of development to be organised around a single product or product family. whereas a separate maintenance group usually looks after a portfolio of different products.

These are issues that should concern maintenance managers when service level agreements are defined. or when they are initially bidding for a maintenance contract.

B. A. KITCHENHAM *ET AL.*

3.5.2.3. Staff Skills. In general, the more skilled the maintenance staff, the better the productivity and quality of maintenance activities. Different activities require different skills. so these factors need to be controlled or specified during empirical studies of maintenance activities.

3.5.3. Customer and user staff

Customer and user issues that affect maintenance are:

- The size of the user population. which affects the amount of work required to support a particular application.
- The variability of the user population, which affects the scope of maintenance tasks. The more varied the user population, the more varied the problems they will encounter and refer to the maintenance staff.
- Whether or not the client and maintenance organisation are part of the same company. Relationships between client and maintenance group may be less co-operative if the groups are from different companies.
- The extent to which the customer/client and users have common goals. Customers/clients fund maintenance activities. If they do not understand the requirements of the real users, they may impose inappropriate service level agreements. to the detriment of the product users who will in turn become less satisfied with the maintenance organisation.

4. TWO MAINTENANCE SCENARIOS

4.1. Organisation distinction

Figure 7 shows the full maintenance ontology. In this section, we use this ontology to specify two different maintenance scenarios. Staff responsibility seems to be one of the most important factors in the above ontology. Our discussion at the WESS workshop continually returned to the issue of whether or not the maintainers and software developers were the same people.

Therefore. in this section, we define two maintenance scenarios based on this distinction:

- Evolutionary development, and
- Independent maintenance organisation.

We show how the factors identified in the ontology differ in the two scenarios. In addition, we consider for each the related industrial concerns.

4.2. Evolutionary development
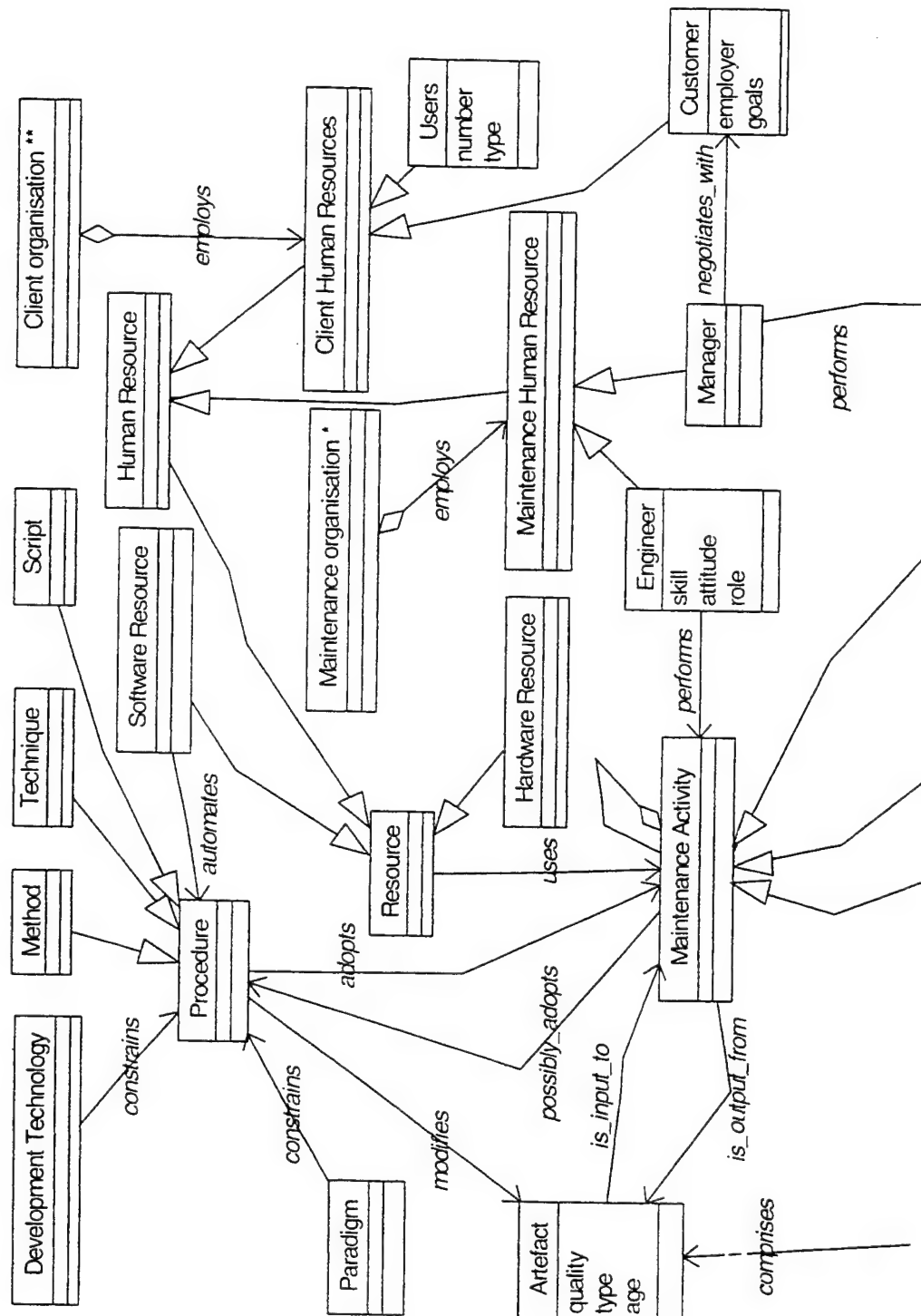
Table 8 specifies the evolutionary development scenario. In this maintenance scenario, practitioners are often concerned with optimising the evolutionary process. Particular concerns include:

- optimisation (and/or minimisation) of inter-release intervals,
- prediction of release quality/reliability,

Copyright © 1999 John Wiley & Sons. Ltd.

J. Softw. Maint: Res. Pract. 11. 365–389 (1999)

Table 8. Evolutionary development scenario

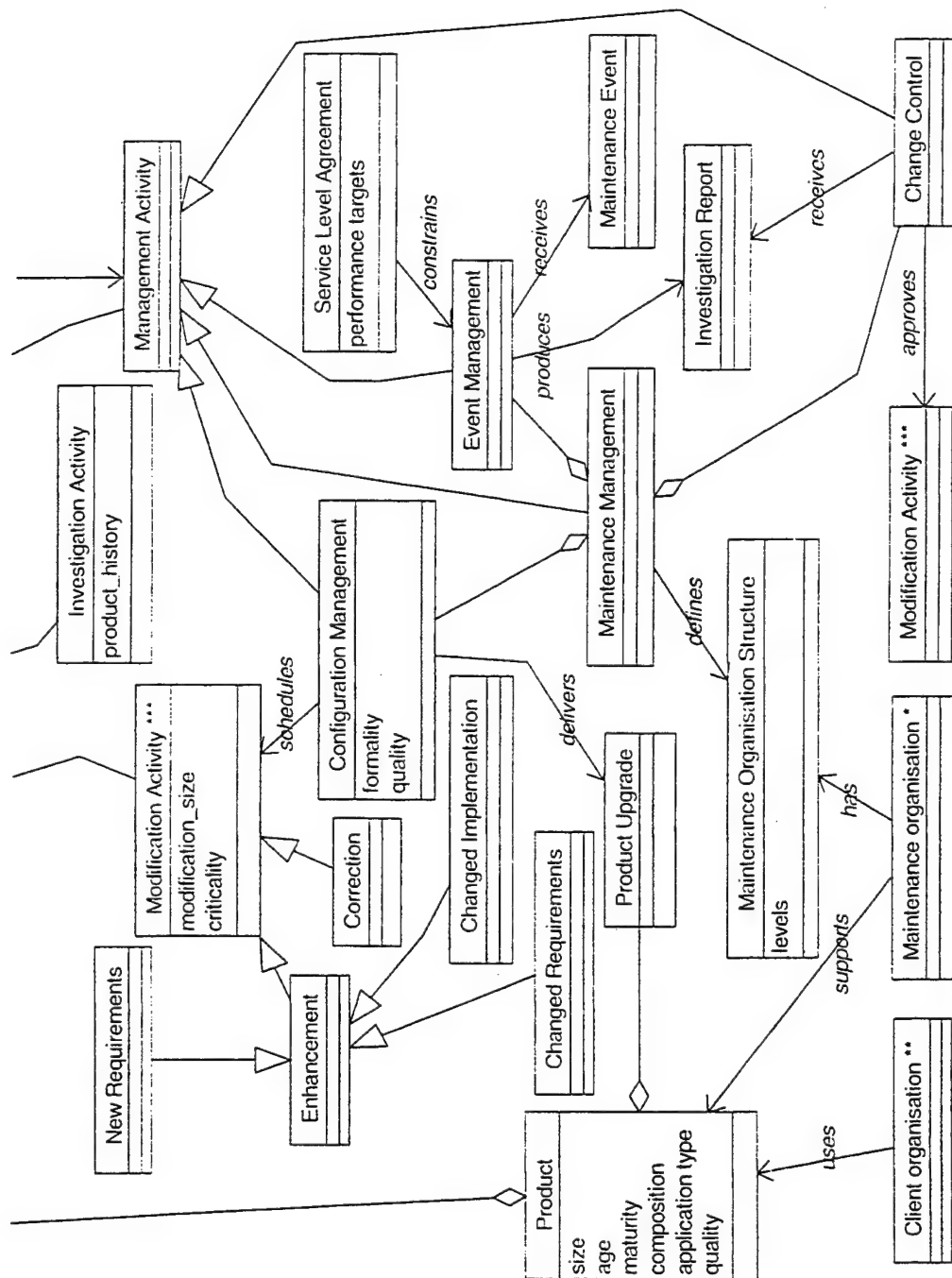|	
Staff responsibilities	Maintenance engineers are responsible both for producing new product upgrades and for correcting problems in past releases. Staff are responsible for the evolution of a single product or product family.
Product size	Usually large. Examples: Space Shuttle, Microsoft Word, ICL VME Operating System. Note, however, large products often encourage small companies to produce small add-on products. These small products track the evolution of larger products. For example PKZIP tools have evolved in line with Microsoft products from DOS to Windows 3.1 to Windows 98.
Development technology	The maintenance and development technologies are identical. Maintenance activities do not require additional staff skills or tools.
Application domain	Application domain knowledge is required both for maintenance and development.
Product age	As the product ages, the original software developers will move to other jobs so some expertise is lost. However, there is also some continuity resulting from the overlap between older staff leaving and new staff joining the group.
Product maturity	The impact of maturity on an evolving product depends on the client and user population. For shrink-wrapped products, there is a danger that maintenance requests arising from a large user population will interfere with enhancement activities. For example, defect reports arising from release n will be received during the development of release $n + 1$. This can be even more complicated if different clients do not upgrade in the same time scale, so some client will be reporting defects with release $n - 2$ while others are reporting problems with release $n - 1$. If one product release is of particularly poor quality, it may generate enough defect reports to prevent software developers working on the next planned release. For custom products, such as the Space Shuttle, releases are co-ordinated with the specific client activities so there is less of a problem.
Maintenance management process	The management will need to provide a means to administer the stream of defect reports from users. Release schedules are based on prioritising customer requirements. Enhancements are funded either by clients (analogous to development projects), or licensing agreements or product sales. Licensing agreements or product sales usually covers maintenance costs.
Maintenance group organisation	Support levels are often used to separate software developers from support staff who interface with users.
Staff attitudes	Staff regard themselves as software engineers rather than developers and maintainers so there are less likely to be problems motivating staff.
Types of maintenance	All enhancement activities are referred to as evolutionary development.
Customer and user types	See product maturity.
Document quality	In principle, the original software documentation would continue to be updated as part of the evolutionary release cycle. However, in practice this would depend on the organisational culture and management practices.

Figure 7. The software maintenance ontology. (The asterisks indicate single classes that are repeated in this Figure to enhance readability)

Table 9. Independent Maintenance Group Scenario

Staff responsibilities	Maintenance engineers are responsible for producing product upgrades that may include changes due to enhancements and corrections of maintenance tasks. They will usually not have been involved in original product development. Staff is usually responsible for a portfolio of products.
Product size	Individual elements in a portfolio will be of different sizes.
Development technology	Usually different products in different portfolios will have been produced using different technologies. The maintenance organisation will often need to support many different technologies although the technologies. required by an individual maintainer will usually be restricted.
Application domain	If the portfolio of products is very diverse. it will be difficult to ensure that all maintenance staff have appropriate domain knowledge.
Product age	Different products will have different ages. This makes the maintenance of portfolios complex and planning and costing maintenance activities difficult.
Product maturity	Different products will have different levels of maturity.
Maintenance management process	The management will need to provide a means to administer the stream of defect reports from users. They need fairly complex estimating and risk management procedures to cope with the complexity inherent in administering portfolios. This will be less formal if the client and maintenance group work for the same company. Relationships with customers are usually mandated by a service agreement. although adaptive maintenance may be managed like a development project.
Maintenance Group Organisation	Support levels are often used to separate software developers from support staff who interface with users.
Staff attitudes	Motivation is likely to be particularly important in maintenance groups.
Types of maintenance	All the standard types of maintenance are performed.
Customer and user types	There seem to be two different scenarios: One client—many users. e.g. in-house support groups. Many Clients—many users. e.g. a third party maintenance shop. Note that in some cases the number of items in the portfolio is important. Some maintenance shops support one large custom product in each client portfolio. e.g. Department of Defense in the U.S.A.
Document quality	This is a critical issue for third party maintenance shops since they seldom have any access to software developers. For in-house support groups it may be less of a problem because they may have access to the original developers.

J. Softw. Maint: Res. Pract. **11**. 365–389 (1999)

128

- effort estimation for individual enhancement projects, and
- planning functional contents of releases to minimise the risk of destabilising the product while achieving customer/client required functionality.

Another important concern is the impact of new development paradigms on system evolution, e.g., RAD products, COTS-based products and object-oriented products.

4.3. Independent maintenance group

Table 9 specifies the independent maintenance group scenario. In this scenario, industry concerns differ according to whether or not the maintenance 'shop' is in-house or a third-party organisation. In particular, third-party organisations have concerns about bidding for maintenance contracts (in terms of estimation processes and accuracy and risks), that are less important for in-house maintenance groups (unless they are candidates for outsourcing). Furthermore, outsourcing organisations—particularly those that takeover in-house organisations—have major management concerns about the issues of achieving a common organisational culture and changing the working methods of organisations they absorb (Tittle, 1998; Ketler and Willems, 1999).

All types of maintenance group have concerns about maintenance task estimating and planning and improving efficiency of maintenance activities. An important issue for such organisations is the need for re-engineering methods and tools to address the problem of lack of adequate specification/design documentation in older products.

5. CONCLUSIONS

This paper has presented an ontology of software maintenance aimed at assisting researchers to report sufficient contextual detail for other researchers and practitioners to understand the results of empirical studies. We developed the ontology from our personal experiences of the maintenance process and have discussed two different maintenance scenarios in terms of the ontology. Figure 7 summarises the ontology, modelled in UML.

One of the problems with the model is that competency questions provide a criterion for inclusion of a factor in the model, but they do not provide completion criteria, nor do they provide any concept of relative importance. Thus, the elements identified in the model are things that a researcher needs to report when describing empirical studies, but there may be other factors we have not included. We must emphasise that, even using this ontology as a guide, it is still the responsibility of the individual researcher to attempt to identify any special conditions that apply to his/her results.

Formally, the ontology presented in this paper is not complete. We have not attempted to formalise the ontology using predicate logic, nor have we fully evaluated it. Furthermore, since we are not attempting to integrate our ontology into a knowledge-based system, we do not believe such a formalisation is necessary. In its current form, we believe the ontology provides useful insights into the type of information researchers should report if we are to understand fully the results of empirical studies of maintenance. Only if the software maintenance community were considering a large-scale database to register empirical research results, would a formalised, fully-evaluated ontology be necessary.

J. Softw. Maint: Res. Pract. 11. 365–389 (1999)

129

References

de Almeida FR, de Menezes SC, da Rocha ARC. 1998. Using ontologies to improve knowledge integration in software engineering environments. In *Proceedings of 2nd World Multiconference on Systemics, Cybernetics and Informatics*, Volume I / *Proceedings of 4th International Conference on Information Analysis and Synthesis*, Volume I: International Institute of Informatics and Systemics: Caracus, Venezuela: pp. 296–304. Also available at URL: http://www.inf.ufes.br/~falbo/download/pub/sci98.zip [28 October 1999].

Fowler M, Scott K. 1997. *UML Distilled: Applying the Standard Object Modeling Language*: Addison-Wesley Publishing Co.: Reading MA.

Gruber TR. 1995. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human–Computer Studies* **43**(5/6):907–928.

Haworth DA, Sharpe S, Hale DP. 1992. A framework for software maintenance: a foundation for scientific inquiry. *Journal of Software Maintenance: Research and Practice* **4**(2):105–117.

Hasselbring W. 1999. Technical opinion: on defining computer science terminology. *Communications of the ACM* **42**(2):88–91.

Karam GM, Casselman RS. 1993. A cataloguing framework for software development methods. *IEEE Computer* **26**(2):34–45.

Ketler K, Willems JR. 1999. A study of the outsourcing decision: preliminary results. In *Proceedings of the 1999 ACM SIG CPR Conference on Computer Personnel Research*: ACM Press: New York NY: pp. 182–189.

Kung H-J, Hsu C. 1998. Software maintenance lifecycle model. In *Proceeding International Conference on Software Maintenance*: IEEE Computer Society Press: Los Alamitos CA: pp. 113–117.

Lehman MM, Perry D, Ramil JF. 1998. Implications of evolution metrics on software maintenance. In *Proceedings International Conference on Software Maintenance*: IEEE Computer Society Press: Los Alamitos CA: pp. 208–217.

Maxwell K, Wassenhove LV, Dutta S. 1996. Software development productivity of European space, military and industrial applications. *IEEE Transactions on Software Engineering* **22**(10):706–718.

Mellor P. 1983. Modelling software support. *ICL Technical Journal* **3**(4):407–438.

Niessink F, Vliet Hv. 1998. Towards mature IT services. *Software Process—Improvement and Practice* **4**(2):55–71.

Pfleeger SL. 1998. *Software Engineering: Theory and Practice*: Prentice-Hall, Inc.: Saddle River NJ: pp. 44–75.

Pressman RS. 1997. *Software Engineering: A Practitioner's Approach*, 4th edition: McGraw-Hill Companies, Inc.: New York: pp. 805–825.

Singer J. 1998. Practises in software maintenance. In *Proceedings International Conference on Software Maintenance*: IEEE Computer Society Press: Los Alamitos CA: pp. 139–145.

Swanson EB, Chapin N. 1995. Interview with E. Burton Swanson. *Journal of Software Maintenance: Research and Practice* **7**(5):303–315.

Tittle J. 1998. Software Maintenance. A Perspective on Some Issues from the Trenches. Keynote Address. *Third Annual Workshop on Empirical Studies of Software Maintenance. WESS '98.* (Slides available at URL: http://www.cs.umd.edu/users/travasso/tittle.ppt [28 October 1999].)

Authors' biographies:

Barbara A. Kitchenham is Managing Director of Butley Software Services Ltd. and Principal Researcher in Software Engineering at the University of Keele. Her main research interests are software metrics and empirical software engineering. She is a visiting professor at the Universities of Bournemouth and Ulster. Her email address is: barbara@cs.keele.ac.uk

Guilherme H. Travassos is an Associate Professor at COPPE-Federal University of Rio de Janeiro. His research interests include empirical software engineering, software architecture, software testing, quality and software engineering environments. He is a Visiting Associate Professor at the Department of Computer Science of the University of Maryland at College Park. His email address is: travassos@cs.umd.edu

Anneliese von Mayrhauser is a Professor at Colorado State University and Director of the Colorado Advanced Software Institute, a consortium of businesses and Colorado universities supporting Technology Transfer research. Her research interests include software testing and maintenance. She holds an M.S. and Ph.D. from Duke University and a Dipl.Inf. from the Karlsruhe Technical University in Germany. Her email address is: avm@cs.colorado.edu

Frank Niessink is a Ph.D. candidate at the Vrije Universiteit, Amsterdam, in the Faculty of Sciences. His research interests are software measurement, software maintenance and process improvement. Frank received an M.Sc. in Computer Science and an M.Sc. in Economics from the Vrije Universiteit. His email address is: F.Niessink@cs.vu.nl

Norman F. Schneidewind is Professor of Information Sciences at the Naval Postgraduate School in Monterey CA. He developed the Schneidewind software reliability model that is used by NASA to predict the software reliability of the Space Shuttle. Also, he is a Fellow of the IEEE, elected for his contributions to software measurement. His email address is: nschneid@nps.navy.mil

Janice Singer is a research officer with the National Research Council in Canada. She works in the Software Engineering Group of the Institute for Information Technology in Ottawa. Her research interests include maintenance, human computer interaction, and empirical studies. She holds a Ph.D. in cognitive psychology from the University of Pittsburgh in Pennsylvania. Her email address is: singer@iit.nrc.ca

Shingo Takada is an Assistant Professor at the Department of Information and Computer Science, Keio University in Japan. His research interests include software engineering, especially software reuse and software maintenance, as well as information exploration. He received his Ph.D. in Computer Science from Keio University in 1995. His email address is: michigan@doi.cs.keio.ac.jp

Risto Vehvilainen is a Managing Director of an IT consulting company. He holds an M.S. in Mathematics from Helsinki University and also he is a doctoral student at the Swedish School of Economics and Business Administration in Helsinki. His main research area is software maintenance as a service. His email address is: risto.vehvilainen@kolumbus.fi

Hongji Yang is a Principal Lecturer in Computer Science Department at De Montfort University, UK and leads the Software Evolution and Re-engineering Group. His research interests include software maintenance, reverse engineering, re-engineering and reuse. He served as a Program Co-Chair at International Conference on Software Maintenance in 1999. His email address is: hjy@dmu.ac.uk

Computer Aided Prototyping System (CAPS) for Heterogeneous Systems Development and Integration[*]

Luqi, V. Berzins, M. Shing, N. Nada and C. Eagle
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
{luqi, berzins, mantak, nnada, cseagle}@cs.nps.navy.mil

Abstract

This paper addresses the problem of how to produce reliable software that is also flexible and cost effective for the DoD distributed software domain. DoD software systems fall into two categories: information systems and war fighter systems. Both types of systems can be distributed, heterogeneous and network-based, consisting of a set of components running on different platforms and working together via multiple communication links and protocols. We propose to tackle the problem using prototyping and a "wrapper and glue" technology for interoperability and integration. This paper describes a distributed development environment, CAPS (Computer-Aided Prototyping System), to support rapid prototyping and automatic generation of wrapper and glue software based on designer specifications. The CAPS system uses a fifth-generation prototyping language to model the communication structure, timing constraints, I/O control, and data buffering that comprise the requirements for an embedded software system. The language supports the specification of hard real-time systems with reusable components from domain specific component libraries. CAPS has been used successfully as a research tool in prototyping large war-fighter control systems (e.g. the command-and-control station, cruise missile flight control system, missile defense systems) and demonstrated its capability to support the development of large complex embedded software.

1. Introduction

DoD software systems are currently categorized into Management Information Systems (MIS) and War Fighter/Embedded Real-time Systems. Both types of systems can be distributed, heterogeneous and network-based, consisting of a set of subsystems, running on different platforms that work together via multiple communication links and protocols. This paper addresses the problem of how to produce reliable software that is also flexible and cost effective for the DoD distributed software system domain, as depicted in the shaded area in Figure 1.
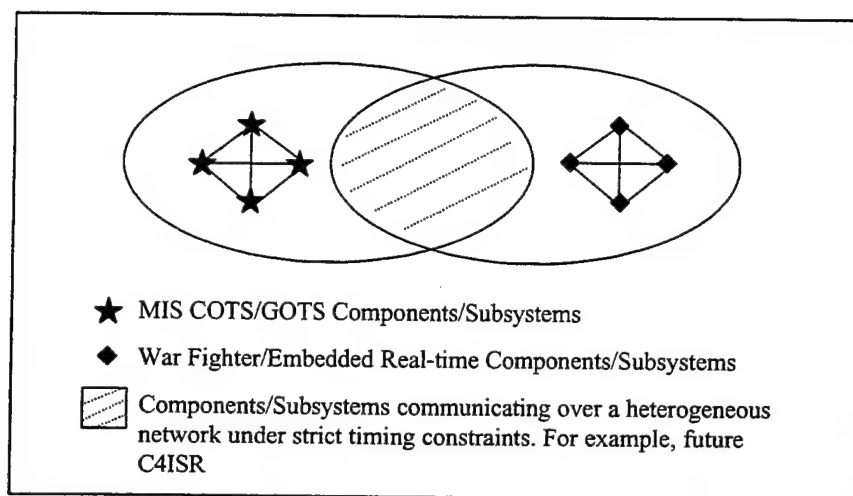
Figure 1. DoD Computer-based systems

Many DoD information systems are COTS/GOTS based (commercial/government off-the-shelf, including "legacy systems"). While using individual COTS/GOTS components saves DoD money, it shifts problems from software development to software integration and interoperability. It is a common belief that interoperability problems are caused by incompatible interface and data formats, and can be fixed "easily" using interface converters and data formatters. However, the real challenges in fixing interoperability problems are incompatible data interpretations, inconsistent assumptions, requirement extensions triggered by global integration issues, and timely data communication between components. Many DoD information systems, especially C4ISR systems, operate under tight timing constraints. Builders of COTS/GOTS based systems have no control over the network on which components communicate. They have to work with available infrastructure and need tools and methods to assist them in making correct design decisions to integrate COTS/GOTS components into a distributed network based system. Similar integration and interoperability problems are common in the commercial sector, and real-time issues are a growing concern. For example, just-in-time manufacturing, on-demand accounting, and factory automation all involve timing requirements. Although software engineers have more control over interfaces and data compatibility between individual components of war fighter systems, they encounter similar data communication problems when they need to connect these components via heterogeneous networks.

We can tackle the problem using prototyping and a "wrapper and glue" technology for interoperability and integration. Our approach is based on a distributed architecture where components collaborate via message passing over heterogeneous networks. It uses a generic interface that allows system designers to specify communication and operating requirements between components as parameters, based on properties of COTS/GOTS components. A separate parameterized model of network characteristics constrains the concrete "glue" software generated for each node. The model enables partial specification of requirements by the system designers, and allows them to explore design alternatives and determine missing parameters via rapid prototyping.

2. The Wrapper and Glue Approach

The cornerstone of our approach is automatic generation of wrapper and glue software based on designer specifications. This software bridges interoperability gaps between individual COTS/GOTS components. Wrapper software provides a common message-passing interface for components that frees developers from the error prone tasks of implementing interface and data conversion for individual components. The glue software schedules time-constrained actions and carries out the actual communication between components. (See Figure 2)
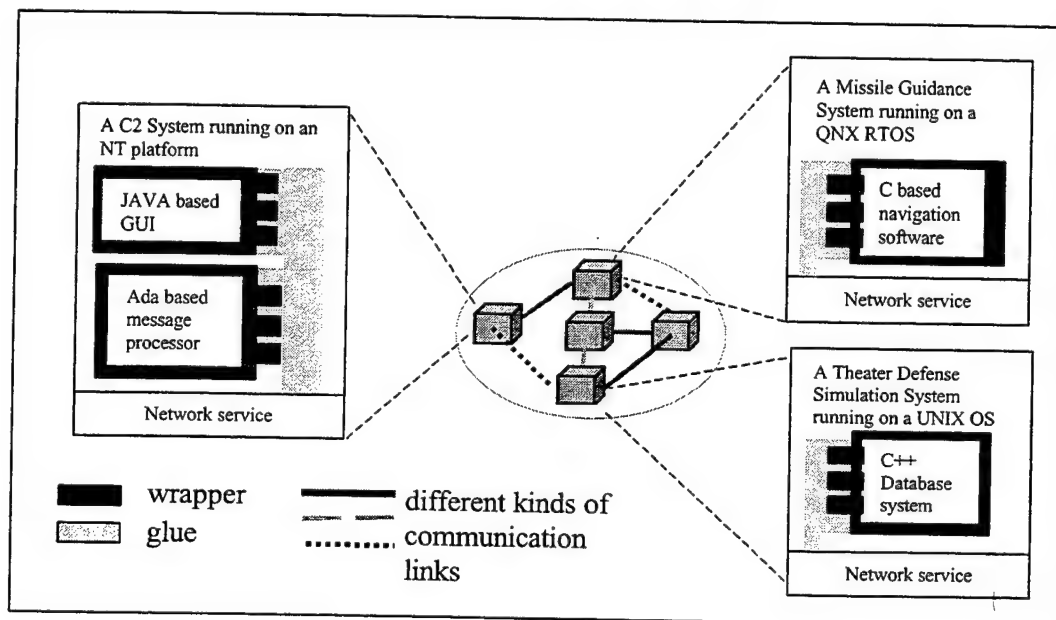
Figure 2. The wrapper and glue software

Our glue-and-wrapper approach uses rapid prototyping and automated software synthesis to improve reliability. It differs from proxy and broker patterns in the object-oriented design literature [4] in that it provides a formal model to support hardware/software co-design. Existing pattern approaches focus on low level data transfer issues. Our approach allows system designers to concentrate on the difficult interoperability problems and issues, while freeing them from implementation details. Prototyping with engineering decision support can help identify and resolve requirements conflicts and semantic incompatibilities.

Glue code works on two levels. It controls the orderly execution of components within a subsystem, and ensures the timely delivery of information between components across a network. Automated generation of glue code depends on automated local and distributed scheduling of actions on heterogeneous computing platforms. Identifying timing constraint conflicts and assessing constraint feasibility are critical in designing and constructing real-time software quickly. Checking whether a set of timing and task precedence constraints can be met on a chosen hardware configuration is known to be a difficult problem. Computer aid is needed in tackling such problem.

134

3. The Computer Aided Prototyping System (CAPS)

The value of computer aided prototyping in software development is clearly recognized. It is a very effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design. Bernstein estimated that for every dollar invested in prototyping, one can expect a $1.40 return within the life cycle of the system development [1]. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply [8]. Computer aid for rapidly and inexpensively constructing and modifying prototypes makes it feasible [10]. The Computer-Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School, is an integrated set of software tools that generate source programs directly from high level requirements specifications [7] (Figure 3). It provides the following kinds of support to the prototype designer:

(1) timing feasibility checking via the scheduler,
(2) consistency checking and automated assistance for project planning, configuration management, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,
(3) computer-aided design completion via the editors,
(4) computer-aided software reuse via the software base, and
(5) automatic generation of wrapper and glue code.

The efficacy of CAPS has been demonstrated in many research projects at the Naval Postgraduate School and other facilities.
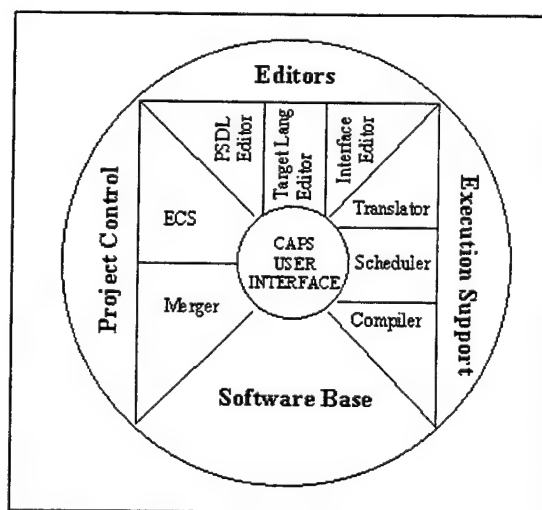

Figure 3. The CAPS Rapid Prototyping Environment

135

3.1 *Overview of the Caps Method*

There are four major stages in the CAPS rapid prototyping process: software system design, construction, execution, and requirements evaluation/modification (Figure 4).
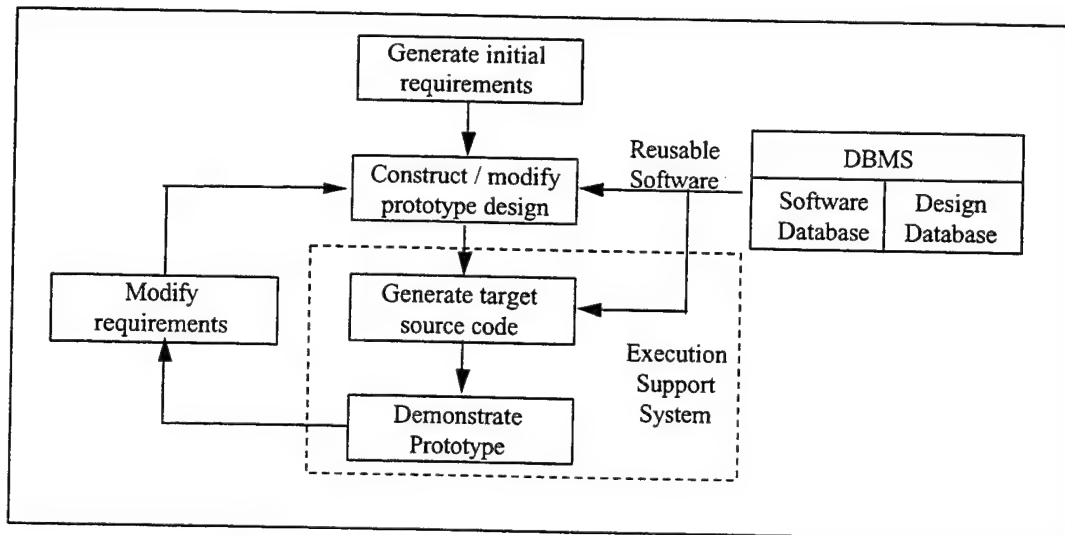
Figure 4. Iterative Prototyping Process in CAPS

The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either informally (e.g. English) or in some formal notation. These requirements may be refined by asking users to verify their completeness and correctness.

After some requirements analysis, the designer uses the CAPS PSDL editor to draw dataflow diagrams annotated with nonprocedural control constraints as part of the specification of a hierarchically structured prototype, resulting in a preliminary, top-level design free from programming level details. The user may continue to decompose any software module until its components can be realized via reusable components drawn from the software base or new atomic components.

This prototype is then translated into the target programming language for execution and evaluation. Debugging and modification utilize a design database that assists the designers in managing the design history and coordinating change, as well as other tools shown in Figure 3.

3.2 *CAPS as a Requirements Engineering Tool*

The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and imprecise, but they are understood best by the customers. The lower levels are more technical, precise, and better suited for the needs of the system analysts and designers, but they are further removed from

136

the user's experiences and less well understood by the customers. Because of the differences in the kinds of descriptions needed by the customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire software development process. CAPS provides the necessary means to bridge the communication gap between the customers and developers. The CAPS tools are based on the Prototype System Description Language (PSDL), which is designed specifically for specifying hard real-time systems [5, 6]. It has a rich set of timing specification features and offers a common baseline from which users and software engineers describe requirements. The PSDL descriptions of the prototype produced by the PSDL editor are very formal, precise and unambiguous, meeting the needs of the system analysts and designers. The demonstrated behavior of the executable prototype, on the other hand, provides concrete information for the customer to assess the validity of the high level requirements and to refine them if necessary.

3.3 *CAPS as a System Testing and Integration Tool*

Unlike throw-away prototypes, the process supported by CAPS provides requirements and designs in a form that can be used in construction of the operational system. The prototype provides an executable representation of system requirements that can be used for comparison during system testing. The existence of a flexible prototype can significantly ease system testing and integration. When final implementations of subsystems are delivered, integration and testing can begin before all of the subsystems are complete by combining the final versions of the completed subsystems with prototype versions of the parts that are still being developed.

3.4 *CAPS as an Acquisition Tool*

Decisions about awarding contracts for building hard real-time systems are risky because there is little objective basis for determining whether a proposed contract will benefit the sponsor at the time when those decisions must be made. It is also very difficult to determine whether a delivered system meets its requirements. CAPS, besides being a useful tool to the hard real-time system developers, is also very useful to the customers. Acquisition managers can use CAPS to ensure that acquisition efforts stay on track and that contractors deliver what they promise. CAPS enables validation of requirements via prototyping demonstration, greatly reducing the risk of contracting for real-time systems.

3.5 *A Platform Independent User Interface*

The current CAPS system provides two interfaces for users to invoke different CAPS tools and to enter the prototype specification. The main interface (Figure 5) was developed using the TAE+ Workbench [11]. The Ada source code generated automatically from the graphic layout uses libraries that only work on SUNOS 4.1.X operating systems. The PSDL editor (Figure 6), which allows users to specify the prototype via augmented dataflow diagram, was implemented in C++ and can only be executed under SUNOS 4.1.X environments. A portable implementation of the CAPS main interface and the PSDL editor was needed to allow users to use CAPS to build PSDL prototypes on different platforms. We choose to overcome these limitations by reimplementing

the main interface (Figure 7) and the PSDL editor (Figure 8) using the Java programming language [2].
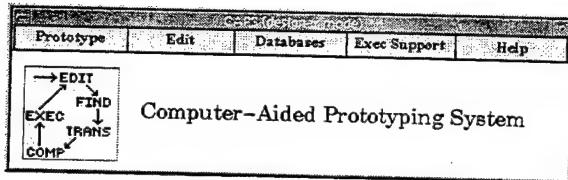
Figure 5. Main Interface of CAPS Release 2.0
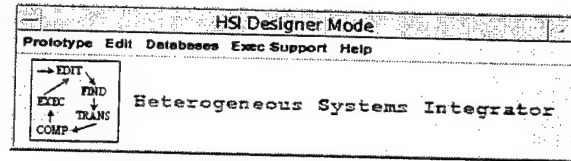
Figure 7. Main Interface of the new CAPS
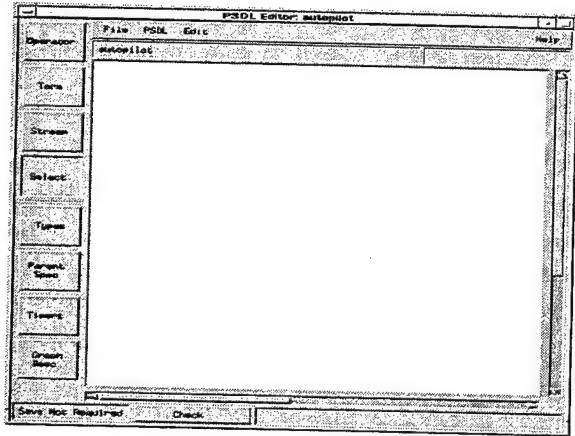
Figure 6. PSDL Editor of CAPS Release 2.0
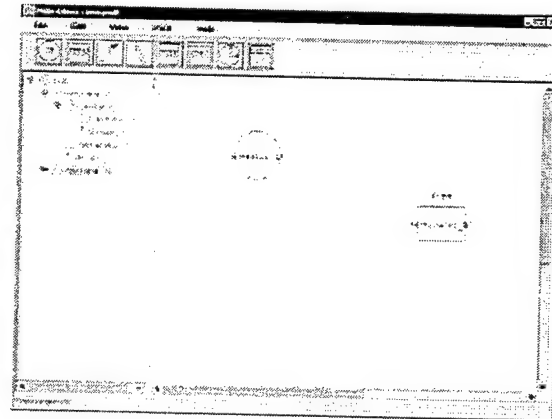
Figure 8. PSDL Editor of the new CAPS

The new graphical user interface, called the Heterogeneous Systems Integrator (HSI), is similar to the previous CAPS. Users of previous CAPS versions will easily adapt to the new interface. There are some new features in this implementation, which do not affect the functionality of the program, but provide a friendlier interface and easier use. The major improvement is the addition of the tree panel on the left side of the editor. The tree panel provides a better view of the overall prototype structure since all of the PSDL components can be seen in a hierarchy. The user can navigate through the prototype by clicking on the names of the components on the tree panel. Thus, it is possible to jump to any level in the hierarchy, which was not possible earlier.

4. A Simple Example: Prototyping a C3I Workstation

To create a first version of a new prototype, users can select "New" from the "Prototype" pull-down menu of the CAPS main interface (Figure 9). The user will then be asked to provide the name of the new prototype (say "c3i_system") and the CAPS PSDL editor will be automatically invoked with a single initial root operator (with a name same as that of the prototype).

138
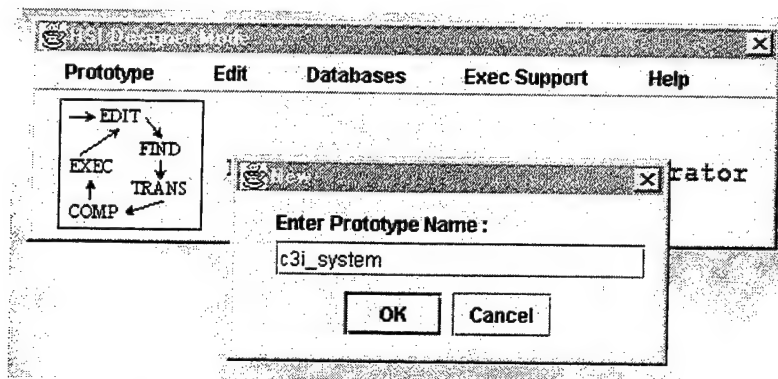
Figure 9. Creating a new prototype called C3I_System

CAPS allows the user to specify the requirements of prototypes as augmented dataflow graphs. Using the drawing tools provided by the PSDL editor, the user can create the top-level dataflow diagram of the c3i_system prototype as shown in Figure 10, where the c3i_system prototype is modeled by nine modules, communicating with each other via data streams. To model the dynamic behavior of these modules, the dataflow diagram is augmented with control and timing constraints. For example, the user may want to specify that the weapons_interface module has a maximum response time of 3 seconds to handle the event triggered by the arrival of new data in the weapon_status_data stream, and it only writes output to the weapon_emrep stream if the status of the weapon_status_data is damage, service_required, or out_of_ammunition. CAPS allow the user to specify these timing and control constraints using the pop-up operator property menu (Figure 11), resulting in a top-level PSDL program shown in Figure 12.

To complete the specification of the c3i_system prototype, the user must specify how each module will be implemented by choosing the implementation language for the module via the operator property menu. The implementation of a module can be in either the target programming language or PSDL. A module with an implementation in the target programming language is called an atomic operator. A module that is decomposed into a PSDL implementation is called a composite operator. Module decomposition can be done by selecting the corresponding operator in the tree-panel on the left side of the PSDL editor.

CAPS supports an incremental prototyping process. The user may choose to implement all nine modules as atomic operators (using dummy components) in the first version, so as to check out the global effects of the timing and control constraints. Then, he/she may choose to decompose the comms_interface module into more detailed subsystems and implement the sub-modules with reusable components, while leaving the others as atomic operators in the second version of the prototype, and so on.
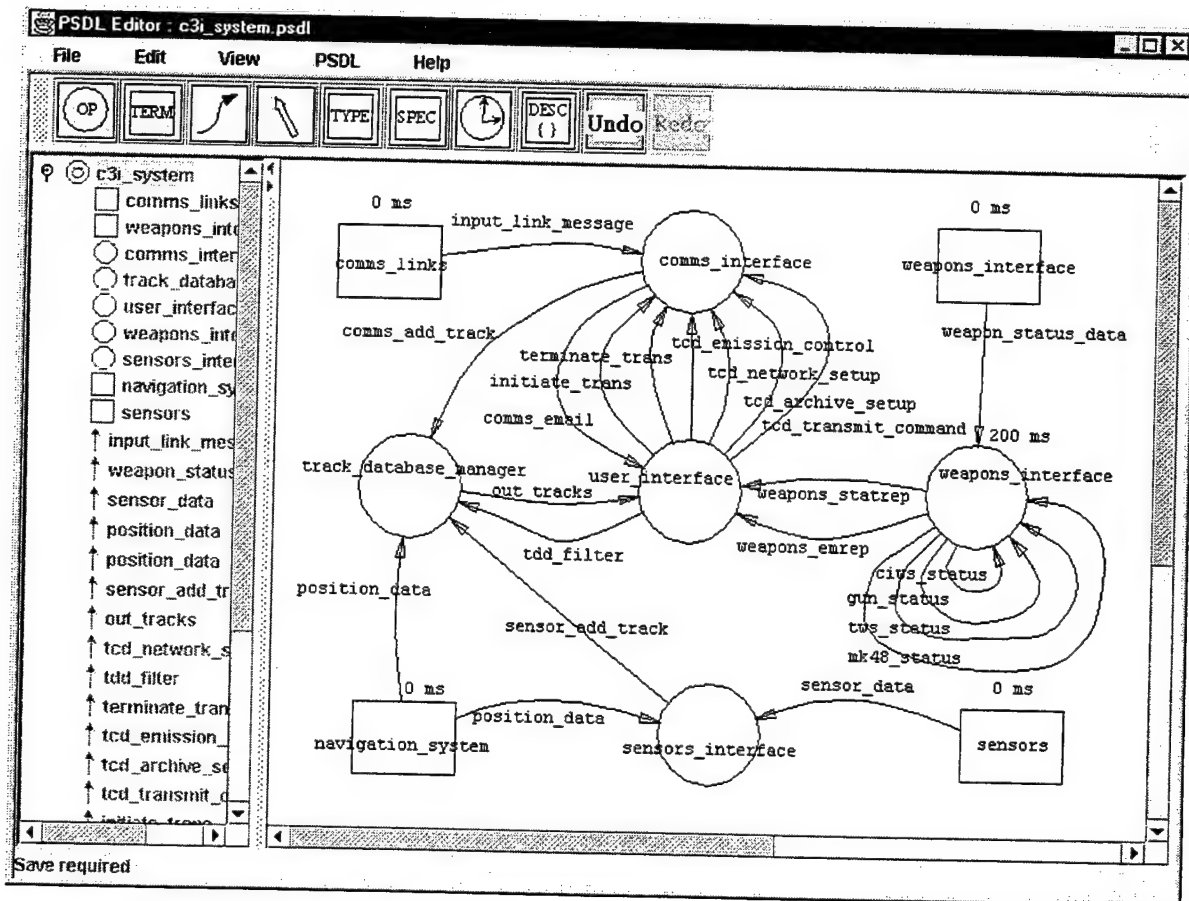
Figure 10. Top-level Dataflow Diagram of the c3i_system.
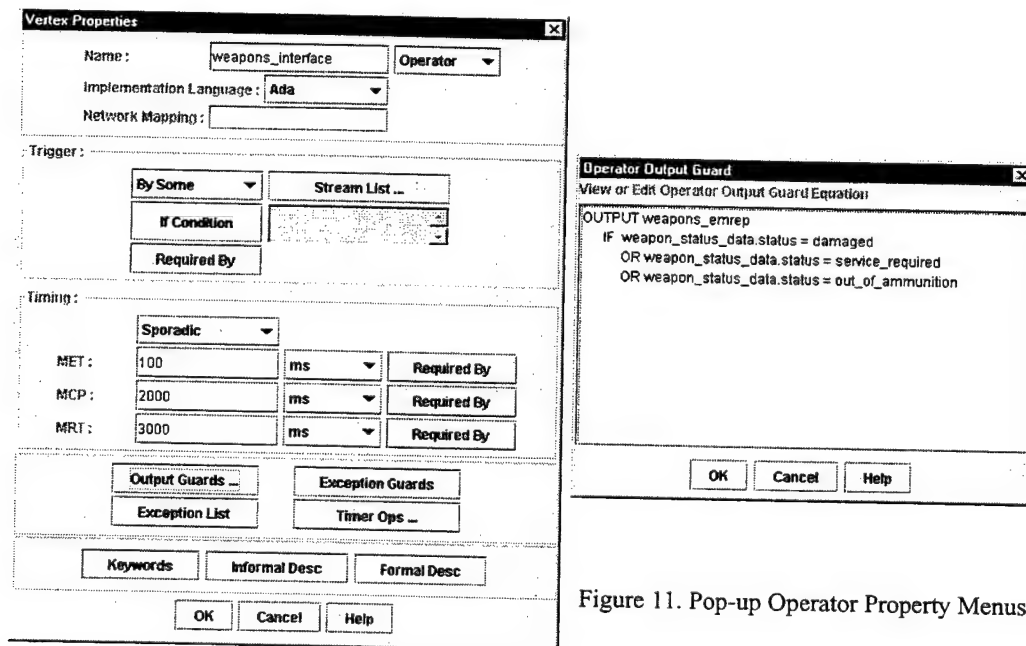
Figure 11. Pop-up Operator Property Menus

```
OPERATOR c3i_system
SPECIFICATION
   DESCRIPTION
      {This module implements a simplified version of
       a generic C3I workstation.}
END
IMPLEMENTATION
   GRAPH
```



```
DATA STREAM
   -- Type declarations for the data streams in the graph go here.
CONTROL CONTRAINTS
   OPERATOR comms_links              OPERATOR weapons_interface
      PERIOD 30000 MS                   TRIGGERED BY SOME
                                           weapon_status_data
   OPERATOR navigation_system        MINIMUM CALLING PERIOD 2000 MS
      PERIOD 30000 MS                MAXIMUM RESPONSE TIME 3000 MS
                                     OUTPUT
   OPERATOR sensors                     weapons_emrep
      PERIOD 30000 MS                   IF weapon_status_data.status =
                                              damaged
   OPERATOR weapons_systems          OR weapon_status_data.status =
      PERIOD 30000 MS                        service_required
                                     OR weapon_status_data.status =
END                                        out_of_ammunition
```

Figure 12. Top-level Specification of the c3i_system

To facilitate the testing of the prototypes, CAPS provides the user with an execution support system that consists of a translator, a scheduler and a compiler. Once the user finishes specifying the prototype, he/she can invoke the translator and the scheduler from the CAPS main interface to analyze the timing constraints for feasibility and to generate a supervisor module for each subsystem of the prototype in the target programming language. Each supervisor module consists of a set of driver procedures that realize all the control constraints, a high priority task (the static schedule) that executes the time-critical operators in a timely fashion, and a low priority dynamic schedule task that executes the non-time-critical operators when there is time available. The supervisor module also contains information that enables the compiler to incorporate all the software components required to implement the atomic operators and generate the binary code automatically. The translator/scheduler also generates the glue code needed for timely delivery of information between subsystems across the target network.

For prototypes which require sophisticated graphic user interfaces, the CAPS main interface provides an interface editor to interactively sculpt the interface. In the c3i_system prototype, we choose to decompose the comms_interface, the track_database_manager and the user_interface modules into subsystems, resulting in hierarchical design consisting of 8 composite operators and twenty-six atomic operators. The user interface of the prototype has a total of 14 panels, four of which are shown in Figure 13. The corresponding Ada program has a total of 10.5K lines of source code. Among the 10.5K lines of code, 3.5K lines comes from supervisor module that was generated automatically by the translator/scheduler and 1.7K lines that were automatically generated by the interface editor [9].

5. Conclusion

CAPS has been used successfully as a research tool in prototyping large war-fighter control systems (e.g. the command-and-control station, cruise missile flight control system, missile defense systems) and demonstrated its capability to support the development of large complex embedded software. Specific payoffs include:

(1) Formulate/validate requirements via prototype demonstration and user feedback
(2) Assess feasibility of real-time system designs
(3) Enable early testing and integration of completed subsystems
(4) Support evolutionary system development, integration and testing
(5) Reduce maintenance costs through systematic code generation
(6) Produce high quality, reliable and flexible software
(7) Avoid schedule overruns

In order to evaluate the benefits derived from the practice of computer-aided prototyping within the software acquisition process, we conducted a case study in which we compared the cost (in dollar amounts) required to perform requirements analysis and feasibility study for the c3i system using the 2167A process, in which the software is coded manually, and the rapid prototyping process, where part of the code is automatically generated via CAPS [3]. We found that, even under very conservative assumptions, using the CAPS method resulted in a cost reduction of $56,300, a 27% cost saving. Taking the results of this comparison, then projecting to a mission control software system, the command and control segment (CCS), we estimated that there would

be a cost saving of 12 million dollars. Applying this concept to an engineering change to a typical component of the CCS software showed a further cost savings of $25,000.

Figure 13. User Interface of the c3i_system

143

6. References

[1] L. Bernstein, "Forward: Importance of Software Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 9-14, 1996.

[2] I. Duranlioglu, *Implementation of a Portable PSDL Editor for the Heterogeneous Systems Integrator*, Master's thesis, Naval Postgraduate School, Monterey, California, March 1999.

[3] M. Ellis, *Computer-Aided Prototyping Systems (CAPS) within the software acquisition process: a case study*, Master's thesis, Naval Postgraduate School, Monterey, California, June 1993.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 1995.

[5] B. Kraemer, Luqi and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Transaction on Software Engineering*, 19(5), pp. 453-477, 1993.

[6] Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transaction on Software Engineering*, 14(10), pp. 1409-1423, 1988.

[7] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, 5(2), pp. 66-72, 1988.

[8] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer*, pp. 111-112, September 1991.

[9] Luqi, "Computer-Aided Prototyping for a Command-and-Control System Using CAPS", *IEEE Software*, 9(1), pp. 56-67, 1992.

[10] Luqi, "System Engineering and Computer-Aided Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 15-17, 1996.

[11] TAE Plus Programmer's Manual (Version 5.1). Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

A Risk Assessment Model for Evolutionary Software Projects[1]

Luqi, J. Nogueira
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

Current early risk assessment techniques rely on subjective human judgments and unrealistic assumptions such as fixed requirements and work breakdown structures. This is a weak approach because different people could arrive at different conclusions from the same scenario even for projects with a stable and well-defined scope, and such projects are rare. This paper introduces a formal model to assess the risk and the duration of software projects automatically, based on objective indicators that can be measured early in the process. The model has been designed to account for significant characteristics of evolutionary software processes, such as requirement complexity, requirement volatility and organizational efficiency. The formal model based on these three indicators estimates the duration and risk of evolutionary software processes. The approach supports (a) automation of risk assessment and, (b) early estimation methods for evolutionary software processes.

1. Introduction

Software applications have grown in size and complexity covering many human activities of importance to society. The report of the President s Information Advisory Committee calls software the new physical infrastructure of the information age . Unfortunately, the ability to build software has not increased proportionately to demand [Hall, 1997. pp xv], and shortfalls in this regard are a growing concern. According to the Standish group, in 1995 84% of software projects finished over time or budget, and $80 billion - $100 billion is spent annually on cancelled projects in the US. Developing software is still a high-risk activity.

There have been many approaches to improving this situation, mostly focused on increasing productivity via improvements in technology or management. Although better productivity is certainly welcome, closer examination shows that these efforts address only half of the problem. A project gets over time or over budget if actual performance does not match estimates. Current estimation techniques are far from reliable, and tend to systematically produce overly optimistic estimates. More accurate early estimates could help reduce wasted resources associated with overruns and cancelled projects in two ways: if costs are known to be too high at the outset, the scope of the project could be reduced to enable completion within time and budget, or it could be cancelled before it starts, and instead the resources could be used to successfully complete other feasible projects.

This paper therefore focuses on improved risk assessment for software projects. We address project risks related to schedule and budget, and focus mostly on completion time of the project. Current risk assessment standards are weak because they rely on subjective human expertise, assume frozen requirements, or depend on metrics difficult to measure until it is too late. This paper describes a formal risk assessment model based on metrics and sensitive to requirements volatility. Further details can be found in [Nogueira 2000]. The model is specially suited for evolutionary prototyping and incremental software development.

Section 2 defines the problem we are addressing. Section 3 analyzes relevant previous work. Section 4 presents and evaluates our project risk model. Section 5 outlines how systematic risk assessment fits into iterative prototyping. Section 6 concludes.

2. The Problem

As the range and complexity of computer applications have grown, the cost of software development has become the major expense of computer-based systems [Boehm 1981], [Karolak 1996]. Research shows that in private industry as well as in government environments, schedule and cost overruns are tragically common [Luqi 1989, Jones 1994, Boehm 1981]. Despite improvements in tools and methodologies, there is little evidence of success in improving the process of moving from the concept to the product, and little progress has been made in managing software development projects [Hall, 1997]. Research shows that 45 percent of all the causes for delayed software deliveries are related to organizational issues [vanGenuchten 1991]. A study published by the Standish Group reveals that the number of software projects that fail has dropped from 40% in 1997 to 26% in 1999. However, the percentage of projects with cost and schedule overruns rose from 33% in 1997 to 46% in 1999 [Reel 1999].

Despite the recent improvements introduced in software processes and automated tools, risk assessment for software projects remains an unstructured problem dependent on human expertise [Boehm 1988, Hall 1997]. The acquisition and development communities, both governmental and industrial, lack systematic ways of identifying, communicating and resolving technical uncertainty [SEI 1996].

This paper explores ways to transform risk assessment into a structured problem with systematic solutions. Constructing a model to assess risk based on objectively measurable parameters that can be automatically collected and analyzed is necessary. Solving the risk assessment problem with indicators measured in the early phases would constitute a great benefit to software engineering. In these early phases, changes can be made with the least impact on the budget and schedule. The requirements phase is the crucial stage to assess risk because: a) it involves a huge amount of human interaction and communication that can be misunderstood and can be a source of errors; b) errors introduced at this phase are very expensive to correct if they are discovered late; c) the existence of software generation tools can diminish the errors in the development process if the requirements are correct; and d) requirements evolve introducing changes and maintenance along the whole life cycle.

Part of the problem is misinterpreting the importance of risk management. It is usually and incorrectly viewed as an additional activity layered on the assigned work, or worse, as an outside activity that is not part of the software process [Hall 1997, Karolak 1996]. One of the goals of our research is to integrate a risk assessment model with previous research on CAPS[2] at NPS [Harn 99]. This integration is required in order to capture metrics automatically in the context of a modern evolutionary prototyping and software development process. This should provide project managers with a more complete tool that can enable improved risk assessment without interfering with the work of a project s software engineers.

A second source of problems in risk management is the lack of tools [Karolak 1996]. The main reason for this lack of tools is that risk assessment is apparently an unstructured problem. To systematize unstructured problems it is necessary to define structured processes. Structured processes involve routine and repetitive problems for which a standard solution exists. Unstructured processes require decision-making based on a three-phase method (intelligence, design, choice) [Turban et al 1998]. An unstructured problem is one in which none of the three phases is structured. Current approaches to risk management are highly sensitive to managers perceptions and preferences, which are difficult to represent by an algorithm. Depending on the decision-maker's attitude towards risk, he or she can decide early with little information, or can postpone the decision, gaining time to obtain more information, but losing some control.

A third source of risk management problems is the confusion created by the informal use of terms. Often, the software engineering community (and most parts of the project management

---

[2] CAPS stands for Computer Aided Prototyping System [Luqi 1988].

community [Wideman 1992]) uses the term "risk" casually. This term is often used to describe different concepts. It is erroneously used as a synonym of "uncertainty" and "threat" [SEI 1996, Hall 1997, Karolak, 1996]. Generally, software risk is viewed as a measure of the likelihood of an unsatisfactory outcome and a loss affecting the software from different points of view: project, process, and product [Hall 1997, SEI 1996]. However, this definition of risk is misleading because it confounds the concepts of risk and uncertainty. In general, most parts of decision-making in software processes are under uncertainty rather than under risk. Uncertainty is a situation in which the probability distribution for the possible outcomes is not known.

In this paper the term "risk" is reserved to indicate the probabilistic outcome of a succession of states of nature, and the term "threat" is used to identify the dangers that can occur. We define risk to be the product of the value of an outcome times its probability of occurrence. This outcome could be either positive (gain) or negative (loss). This abstraction permits one to address not only the classical risk management issue, but also to discover opportunities leading to competitive advantage.

We address the issue of risk assessment by estimating the probability distribution for the possible outcomes of a project, based on observed values of metrics that can be measured early in the process. The metrics were chosen based on a causal analysis to identify the most important threats and a statistical analysis to choose the shape of the probability distribution and relate its parameters to readily measurable metrics.

3. Related Work

There are three main groups of research related to risk:

- **Assessing Software Risk by Measuring Reliability.** This group follows a probabilistic approach and has successfully assessed the reliability of the product [Lyu 1995, Schneidewind 1975, Musa 1998]. However, this approach addresses the reliability of the product, not the risk of failing to complete the project within budget and schedule constraints. These approaches could be used to assess risks related to failures of software projects, which are outside the scope of the current paper. A concern with these approaches is that the resulting assessments arrive too late to economically correct possible faults, because the software product is mostly complete and development resources are mostly gone at the time when reliability of the product can be assessed by testing.

- **Heuristic approaches:** Other researchers assess the risk from the beginning, in parallel with the development process. However, these approaches are less rigorous, typically subjective and weakly structured. Basically these approaches use lists of practices and checklists [SEI, 1996, Hall 1997, Charette 1997, Jones 1994] or scoring techniques [Karolak 1996]. Paradoxically, SEI defines software technical risk as a measure of the probability and severity of adverse effects in developing software that does not meet its intended functions and performance requirements [SEI, 1996]. However, the term "probability" is misleading in this case because the probability distribution is unknown.

- **Macro Model Approaches:** A third group of researchers uses well known estimation models to assess how risky a project could be. The widely used methods COCOMO [Boehm 1981], and SLIM [Putnam, 1980] both assume that the requirements will remain unchanged, and require an estimation of the size of the final product as input for the models [Londeix 1987]. This size cannot be actually measured until late in the project.

The standard tools used to control all types of projects, including PERT, CPM, and Gantt, do not consider coordination and communication overhead. Such models represent sequential interdependencies through explicit representation of precedence relationships between activities. This simplified vision of a project cannot address the dynamics created by reciprocal requirements of information in concurrent activities, exception management, and the impact of

actor interactions. Since the missing factors increase time requirements, the estimates resulting from these generic project estimation models are overly optimistic.

These issues are addressed by Vit Project [Levitt 1999, Thomsen et al. 1999]. Vit Project is applicable to projects in which a) all activities in the project can be predefined; b) the organization is static, and all activities are pre-assigned to actors in the static organization; c) the exceptions to activities result in extra work volume for the predefined activities and are carried out by the pre-assigned actors; and d) actors are assumed to have congruent goals. The model is well suited for simulating organizations that deal with great amounts of information processing and coordination. Such characteristics are extremely relevant in software processes [Boehm, 1981]. However, this approach requires a fixed work breakdown structure, and therefore does not apply at the early stages when requirements are changing and the set of tasks comprising the project are still uncertain.

By using informal risk assessment models, using estimation models based on optimistic assumptions that require parameters difficult to provide until late, and using optimistic project control tools, project managers condemn themselves to overrun schedules and cost.

4. The Proposed Project Risk Model

Our approach is based on metrics automatically collectable from the engineering database from near the beginning of the development. The indicators used are Requirements Volatility (RV), Complexity (CX), and Efficiency (EF).

Requirement Volatility (RV): RV is a measure of three characteristics of the requirements: a) the Birth-Rate (BR), that is the percentage of new requirements incorporated in each cycle of the evolution process; b) the Death-Rate (DR), that is the percentage of requirements dropped in each cycle; and c) the Change-Rate (CR) defined as the percentage of requirements changed from the previous version. A change in one requirement is modeled as a birth of a new requirement and the death of another, so that CR is included in the measured values of BR and DR. RV is calculated as follows: $RV = BR + DR$.

Complexity (CX): Complexity of the requirements is measured from a formal specification. A requirements representation that supports computer-aided prototyping, such as PSDL [Luqi 1996], is useful in the context of evolutionary prototyping. We define a complexity metric called Large Granularity Complexity (LGC) that is calculated as follows: $LGC = O + D + T$, where for PSDL O is the number of atomic operators (functions or state machines), D is the number of atomic data streams (data connections between operators), and T is the number of abstract data types required for the system. Operators and data streams are the components of a dataflow graph. This is a measure of the complexity of the prototype architecture, similar in spirit to function points but more suitable for modeling embedded and real-time systems. The measure can also be applied to other modeling notations that represent modules, data connections, and abstract data types or classes. We found a strong correlation between the complexity measured in LGC and the size of PSDL specifications (correlation coefficient $R = 0.996$). Most important, we also found a strong correlation ($R = 0.898$) between the complexity measured in LGC and the size of the final product expressed in non-comment lines of Ada code, including both the code automatically created by the generator and the code manually introduced by the programmers.

Efficiency (EF): The efficiency of the organization is measured using a direct observation of the use of time. EF is calculated as a ratio between the time dedicated to direct labor and the idle time: $EF = Direct\ Labor\ Time\ /\ Idle\ Time$. We found that this easily measurable quantity was a good discriminator between high team productivity and low team productivity in a set of simulated software projects [Nogueira 2000].

We validated and calibrated our model with a series of simulated software projects using Vit Project. This tool was chosen because of the inclusion of communications and exceptions in its project dynamics model, and because it has been extensively validated for many types of engineering projects, including software engineering projects. The input parameters for the simulated scenarios were RV, EF and CX, and the observed output was the development time. Given that the proposed model uses parameters collected during the early phases and given that Vit Project requires a complete breakdown structure of the project, which can be done only in the late phases, there was a considerable time gap between the two measurements. This time gap is less than for a post-mortem analysis, but it is sufficient for model calibration and validation purposes.

The simulation results were analyzed statistically, with the finding that the Weibull probability distribution was the best fit for all the samples. A random variable x is said to have a Weibull distribution with parameters α, β and γ (with $\alpha > 0$, $\beta > 0$) if the probability distribution function (pdf) and cumulative distribution function (cdf) of x are respectively:

$$\text{pdf: } f(x; \alpha, \beta, \gamma) = \begin{cases} 0, & x < \gamma \\ (\alpha/\beta^\alpha)(x-\gamma)^{\alpha-1}\exp(-((x-\gamma)/\beta)^\alpha), & x \geq \gamma \end{cases}$$

$$\text{cdf: } F(x; \alpha, \beta, \gamma) = \begin{cases} 0, & x < \gamma \\ 1 - \exp(-((x-\gamma)/\beta)^\alpha) & x \geq \gamma. \end{cases}$$

The random variable under study, x, can be interpreted as development time in our context. The shape parameter α controls the skew of the pdf, which is not symmetric. We found that this is mostly related to the efficiency of the organization (EF). The scale parameter β stretches or compresses the graph in the x direction. We found that this parameter is related to the efficiency (EF), requirements volatility (RV), and complexity (CX) measured in LGC. The shifting parameter γ is shifts the origin of the curves to the right. We found that it is mostly related to the complexity measured in LGC.

Based on best fit to our simulation results, the model parameters can be derived from the project metrics using the following algorithm:

```
If (EF > 2.0)    then  α = 1.95;
                       γ = 22 * 0.32*(13*ln(LGC)—82);
                       β = γ/(5.71+(RV-20)*0.046);
                 else  α = 2.5;
                       γ = 22 * 0.85*(13*ln(LGC)—82);
                       β = γ /(5.47-(RV-20)*0.114);
                 end if;
```

The model estimates the following cumulative probability distribution for project completion on or before time x:

```
P(x) = 1 - exp(-(((x - γ)/β)^α)) // where x is time in days
```

This equation can be inverted to obtain the schedule length needed to have a probability P of completing within schedule, with the following result.

```
x = γ + β(-ln(1—P))^(1/α)
```

The probability P can be interpreted as a degree of confidence in the ability of the project to successfully complete within a schedule of length x. Applying the above equation to estimate the development time needed for a 95% chance of completion within schedule for 16 different 149

scenarios simulated using Vit Project, we observed a standard error of 22 days. The worst case was an error of 60 days for a project of 520 days (12%). The comparison of estimated time and simulated time is shown below.
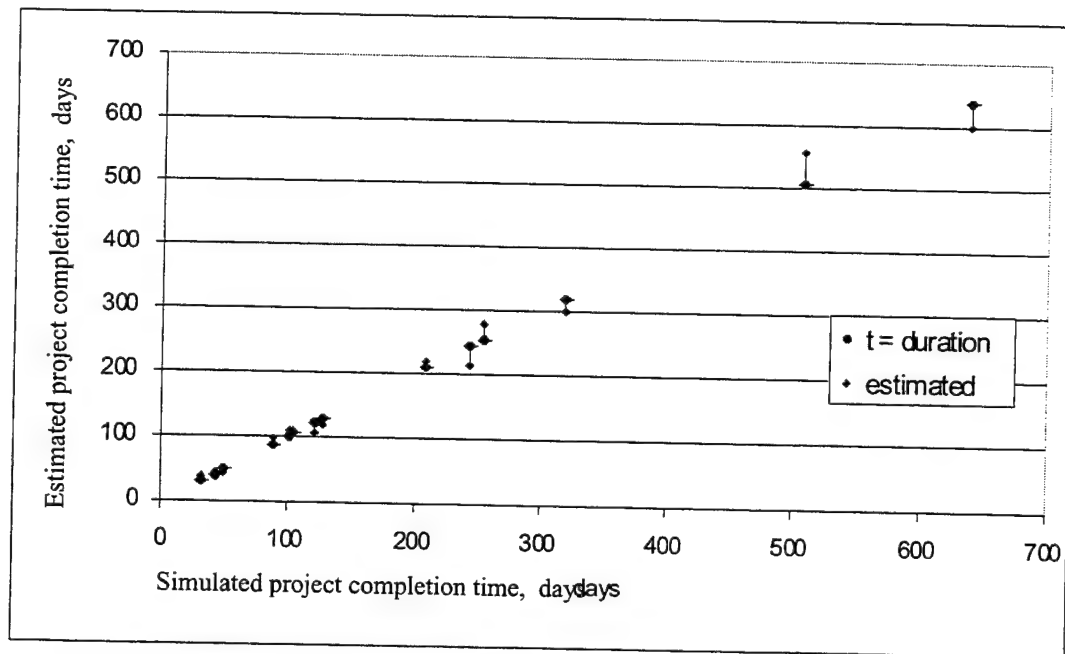
5. Integrating Risk Assessment into Prototyping

The model presented in the previous section is designed to support an iterative prototyping and software development process. In this process, an initial problem statement, a prototype demo or problem reports from a deployed software product trigger an issue analysis, followed by formulation of proposed requirements changes, and specification of a proposed adjustment to the software requirements, which can be initially empty. At this point in each cycle, the project manager should perform a risk assessment step. The results of the risk assessment step guide the degree of detail to which requirements enhancements are demonstrated, and the set of requirements issues to be considered in the next prototyping cycle, if any.

150
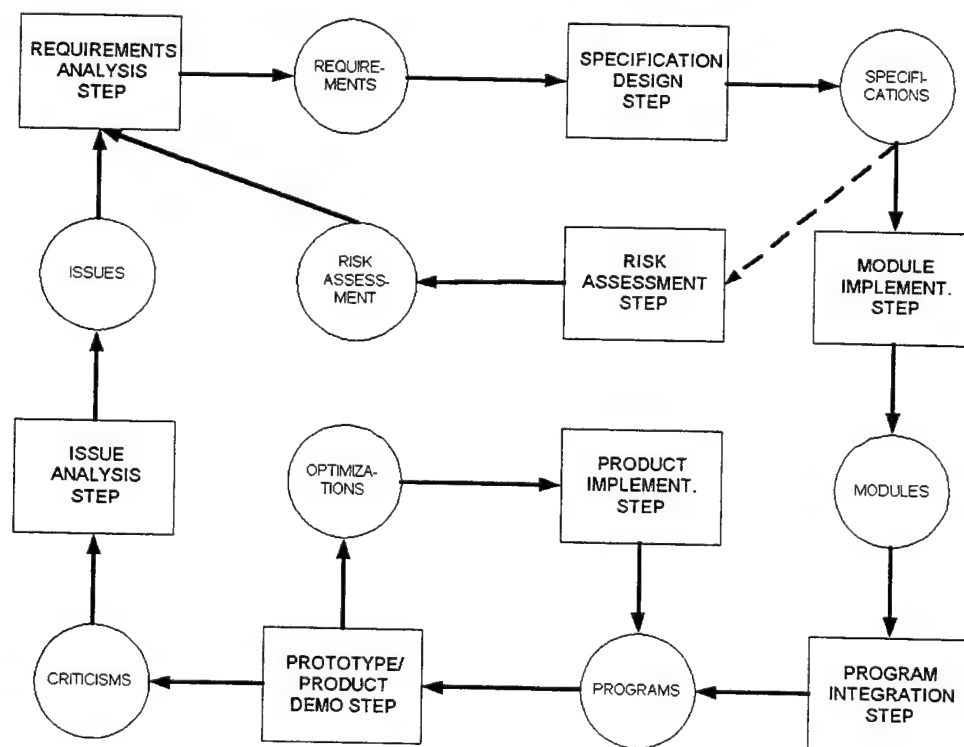
The first measurement-based risk assessment step can be performed after specification of the first version of the prototype architecture, based on the requirements volatility, LGC and efficiency measurements from the steps just performed.

In cases where risk assessments are required even earlier, before any prototyping has been done, estimates of team efficiency and requirements volatility can be based on measurements of similar past projects, and initial complexity estimates can be based on subjective guesswork of the kind currently used in the macro model approaches. This kind of estimate may be less reliable than those based solely on measurements, but it can provide a principled and reasonably accurate basis for deciding whether or not to start a prototyping process to determine the requirements for a proposed development project. Thus parts of our approach can be used truly at the very beginning of the process.

If a prototyping effort is approved, early measurements of the process could be used to refine the initial estimates of the model parameters using Bayesian methods, thus providing a balanced and systematic transition from subjective guesswork, coded as an *a priori* distribution, to assessments increasingly based on systematic measurement. Such an approach also supports incorporation and systematic refinement of measurements from previous cycles of the iterative prototyping process.

The results of risk assessment can provide guidance on the degree to which the project can afford to explore requirements enhancements requested by the customers. It can also help customers or marketing departments to decide how much they really want possible improvements, in the context of the resulting time and cost estimates. Systematic cost/benefit analysis becomes possible only with the availability of reasonably accurate estimates.

The risk assessment step can thus provide a balancing force to stabilize the requirements formulation process. In the absence of information on how much potential enhancements will cost, stakeholders are prone to unrealistic requirements amplification — of course they would always like to have a better system, no matter how good the existing one is, if you do not ask them to pay for the improvements. The proposed risk assessment steps can provide a realistic basis for incorporating time and cost constraints and cost/benefit tradeoffs early in the process, when the situation is fluid and many options are open.

This process refinement provides some additional insight into the dynamics of iterative prototyping: the iterative process should stop when the customers have determined what requirements they can afford to realize, and which of many possible improvements they will be willing to pay for, if any. It is not necessarily the case that the set of criticisms elicited by the final round of prototype demonstrations is empty — that is true only in an idealized world with adequate budgets and patient customers.

6. Conclusion

This paper introduces a formal risk assessment model for software projects based on probabilities and metrics automatically collectable from the project baseline. The approach enables a project manager to evaluate the probability of success of the project very early in the life cycle, during an iterative requirements formulation process, based on well-defined measurements rather than just guesswork or subjective judgments.

For more than twenty years, estimation standards have been characterized by a common limitation: the requirements should be frozen in order to make estimates. This model presented in this paper removes this important limitation, facing the reality that requirements are inherently variable.

The model is perfectly suited for any evolutionary software process because it follows the same philosophy. The risk assessment and estimation steps are conducted at each evolutionary cycle with increasing knowledge and decreasing variance. The research formalizes an

improvement in the evolutionary software process, introducing a risk assessment step that can be automated, and that can help shape the planning of the project in the early stages when there is still substantial freedom to allocate available time and budget.

References

[Boehm 1981]   B. Boehm, *Software Engineering Economics,* Prentice Hall, 1981.

[Boehm 1988]   B. Boehm, A Spiral Model of Software Development and Enhancement, *Computer,* May 1988.

[Charette 1997]   R. Charette, K. Adams, & M. White, Managing Risk in Software Maintenance, *IEEE Software,* May-June, 1997.

[Gilb 1977]   T. Gilb, *Software Metrics,* Winthrop Publishers, Inc., 1977.

[Hall 1997]   E. Hall, Managing Risk, *Methods for Software Systems Development,* Addison Wesley, 1997.

[Harn 1999]   M. Harn, V. Berzins, Luqi, Computer-Aided Software Evolution Based on a Formal Model, *Proceedings of the Thirteenth International Conference on Systems Engineering,* Las Vegas, Nevada, August 9-12, 1999, pp. CS: 55-60.

[Jones 1994]   C. Jones, *Assessment and Control of Software Risks,* Yourdon Press Prentice Hall, 1994.

[Karolak 1996]   D. Karolak, Software *Engineering Management,* IEEE Computer Society Press, 1996.

[Levitt 1999]   R. Levitt, *The ViteProject Handbook: A User's Guide to Modeling and Analyzing Project Work Processes and Organizations,* Vit ' 1999.

[Londeix 1987]   B. Londeix, *Cost Estimation for Software Development,* Addison-Wesley, 1987.

[Luqi 1988]   Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software,* Vol. 5, No. 2, p. 66-72, March 1988.

[Luqi 1989]   Luqi, Software Evolution Through Rapid Prototyping, *IEEE Computer,* May 1989.

[Luqi 1996]   Luqi, Special Issue: Computer-Aided Prototyping, *Journal of Systems Integration,* Vol. 6, Nos. 1-2, March 1996.

[Lyu 1995]   M. Lyu, *Software Reliability Engineering,* IEEE Computer Society Press. 1995.

[Musa 1998]   J. Musa, Software *Reliability Engineering: More Reliable Software, Faster Development and Testing,* McGraw-Hill, 1998.

[Nogueira 2000]   J. Nogueira, *A Formal Risk Assessment Model for Software Projects,* Ph.D. Dissertation, Naval Postgraduate School, 2000.

[Putnam 1980]   L. Putnam, *Software Cost Estimating and Life-cycle Control: Getting the Software Numbers,* IEEE Computer Society Press, 1980.

[Reel 1999]   J. Reel, *Critical Success Factors in Software Projects,* IEEE Software, May - June 1999.

[SEI 1996]   Software Engineering Institute, Software Risk Management, Technical Report CMU/SEI-96-TR-012, June 1996.

[Schneidewind 1975] N. Schneidewind, Analysis of Error Processes in Computer Software, *Proceedings of the International Conference on Reliable Software,* IEEE Computer Society, 21-23 April 1975, p 337-346.

[Turban et al 1998] E. Turban and J. Aronson, *Decision Support Systems and Intelligent Systems,* Prentice Hall, 1998.

[vanGenuchten 1991] M. van Genuchten, Why is Software Late? An Empirical Study of the Reasons for Delay in Software Development, *IEEE Transactions on Software Engineering,* June, 1991.

[Wideman 1992]   R. Wideman, *Risk Management: A Guide to Managing Project Risk Opportunities,* Project Management Institute, 1992.

AUTOMATED PROTOTYPING TOOL-KIT (APT)

N. Nada, V. Berzins, and Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943,
{nnada, Berzins, luqi}@cs.nps.navy.mil
Ph. 831-656-4075
Fax 8310656-3225

Abstract

APT (Automated Prototyping Tool-Kit) is an integrated set of software tools that generate source programs directly from real-time requirements. The APT system uses a fifth-generation prototyping language to model the communication structure, timing constraints, I/O control, and data buffering that comprise the requirements for an embedded software system. The language supports the specification of hard real-time systems with reusable components from domain specific component libraries. APT has been used successfully as a research tool in prototyping large war-fighter control systems (e.g. the command-and-control station, cruise missile flight control system, patriot missile defense systems) and demonstrated its capability to support the development of large complex embedded software.

Keywords: APT, Automated Prototyping, Real-Time Systems, Command and Control, Formal Methods, Evolution, Reuse, Architecture, Components, PSDL

1 INTRODUCTION

Software project managers are often faced with the problem of inability to accurately and completely specify requirements for real-time software systems, resulting in poor productivity, schedule overruns, unmaintainable and unreliable software. APT is designed to assist program managers to rapidly evaluate requirements for military real-time control software using executable prototypes, and to test and integrate completed subsystems through evolutionary prototyping. APT provides a capability to quickly develop functional prototypes to verify feasibility of system requirements early in the software development process. It supports an evolutionary development process that spans the complete life-cycle of real-time software.

2 THE AUTOMATED PROTOTYPING TOOL-KIT (APT)

The value of computer aided prototyping in software development is clearly recognized. It is a very effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design. Bernstein estimated that for every dollar invested in prototyping, one can expect a $1.40 return within the life cycle of the system development [1]. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply [8]. Computer aid for rapidly and inexpensively

constructing and modifying prototypes makes it feasible [10]. The Automated Prototyping Tool-kit (APT), a research tool developed at the Naval Postgraduate School, is an integrated set of software tools that generate source programs directly from high level requirements specifications [7] (Figure 1).

It provides the following kinds of support to the prototype designer:

(1) timing feasibility checking via the scheduler,
(2) consistency checking and automated assistance for project planning, configuration management, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,
(3) computer-aided design completion via the editors,
(4) computer-aided software reuse via the software base, and
(5) automatic generation of wrapper and glue code.

The efficacy of APT has been demonstrated in many research projects at the Naval Postgraduate School and other facilities.
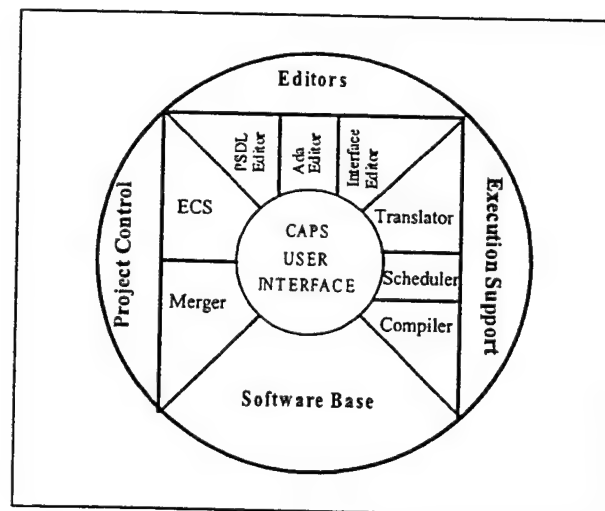
Figure 1. The APT Rapid Prototyping Environment

2.1 *Overview of the APT Method*
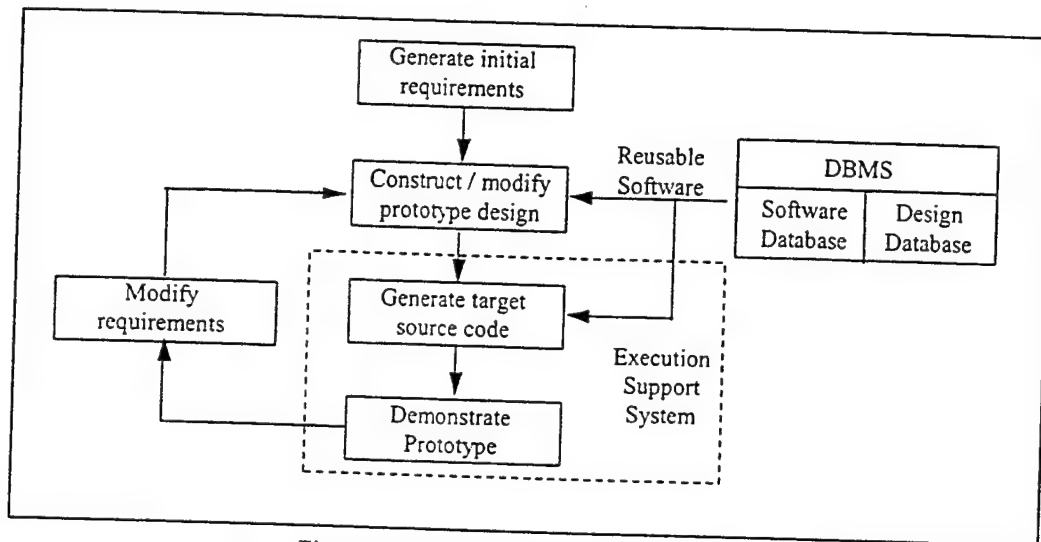
Figure 2. Iterative Prototyping Process in APT

There are four major stages in the APT rapid prototyping process: software system design, construction, execution, and requirements evaluation and/or modification (Figure 2).

The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either informally (e.g. English) or in some formal notation. These requirements may be refined by asking users to verify their completeness and correctness.

After some requirements analysis, the designer uses the APT PSDL editor to draw dataflow diagrams annotated with nonprocedural control constraints as part of the specification of a hierarchically structured prototype, resulting in a preliminary, top-level design free from

programming level details. The user may continue to decompose any software module until its components can be realized via reusable components drawn from the software base or new atomic components.

This prototype is then translated into the target programming language for execution and evaluation. Debugging and modification utilize a design database that assists the designers in managing the design history and coordinating change, as well as other tools shown in Figure 3.

2.2 *APT as a Requirements Engineering Tool*

The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and

155

imprecise, but they are understood best by the customers. The lower levels are more technical, precise, and better suited for the needs of the system analysts and designers, but they are further removed from the user's experiences and less well understood by the customers. Because of the differences in the kinds of descriptions needed by the customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire software development process. APT provides the necessary means to bridge the communication gap between the customers and developers. The APT tools are based on the Prototype System Description Language (PSDL), which is designed specifically for specifying hard real-time systems [5, 6]. It has a rich set of timing specification features and offers a common baseline from which users and software engineers describe requirements. The PSDL descriptions of the prototype produced by the PSDL editor are very formal, precise and unambiguous, meeting the needs of the system analysts and designers. The demonstrated behavior of the executable prototype, on the other hand, provides concrete information for the customer to assess the validity of the high level requirements and to refine them if necessary.

2.3 *APT as a System Testing and Integration Tool*

Unlike throw-away prototypes, the process supported by APT provides requirements and designs in a form that can be used in construction of the operational system. The prototype provides an executable representation of system requirements that can be used for comparison during system testing. The

existence of a flexible prototype can significantly ease system testing and integration. When final implementations of subsystems are delivered, integration and testing can begin before all of the subsystems are complete by combining the final versions of the completed subsystems with prototype versions of the parts that are still being developed.

2.4 *APT as an Acquisition Tool*

Decisions about awarding contracts for building hard real-time systems are risky because there is little objective basis for determining whether a proposed contract will benefit the sponsor at the time when those decisions must be made. It is also very difficult to determine whether a delivered system meets its requirements. APT, besides being a useful tool to the hard real-time system developers, is also very useful to the customers. Acquisition managers can use APT to ensure that acquisition efforts stay on track and that contractors deliver what they promise. APT enables validation of requirements via prototyping demonstration, greatly reducing the risk of contracting for real-time systems.

2.5 *A Platform Independent User Interface*

The current APT system provides two interfaces for users to invoke different APT tools and to enter the prototype specification. The main interface (Figure 3) was developed using the TAE+ Workbench [11]. The Ada source code generated automatically from the graphic layout uses libraries that only work on SUNOS 4.1.X operating systems. The PSDL editor (Figure 4), which allows users to specify the prototype via augmented dataflow diagram, was

implemented in C++ and can only be executed under SUNOS 4.1.X environments. A portable implementation of the APT main interface and the PSDL editor was needed to allow users to use APT to build PSDL prototypes on different platforms. We choose to overcome these limitations by reimplementing the main interface (Figure 5) and the PSDL editor (Figure 6) using the Java programming language [2].
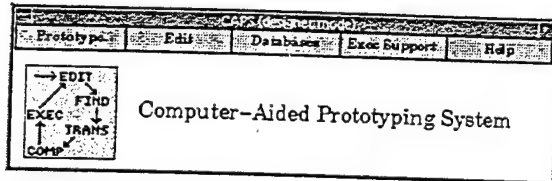
Figure 3. Main Interface of APT Release 2.0
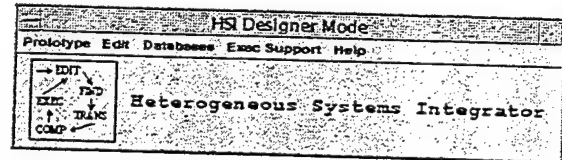
Figure 5. Main Interface of the new APT
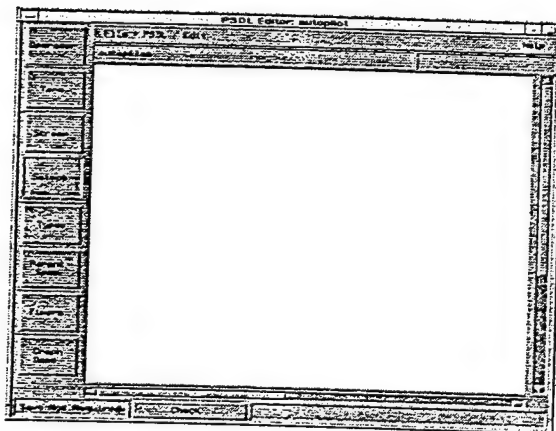
Figure 4. PSDL Editor of APT Release 2.0
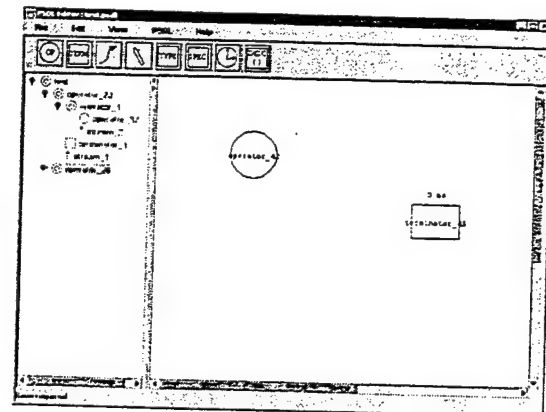
Figure 6. PSDL Editor of the new APT

The new graphical user interface, called the Heterogeneous Systems Integrator (HSI), is similar to the previous APT. Users of previous APT versions will easily adapt to the new interface. There are some new features in this implementation, which do not affect the functionality of the program, but provide a friendlier interface and easier use. The major improvement is the addition of the tree panel on the left side of the editor. The tree panel provides a better view of the overall prototype structure since all of the PSDL components can be seen in a hierarchy. The user can navigate through the prototype by clicking on the names of the components on the tree panel. Thus, it is possible to jump to any level in the hierarchy, which was not possible earlier.

3 A SIMPLE EXAMPLE: PROTOTYPING A C3I WORKSTATION

157

To create a first version of a new prototype, users can select "New" from the "Prototype" pull-down menu of the APT main interface (Figure 7). The user will then be asked to provide the name of the new prototype (say "c3i_system") and the APT PSDL editor will be automatically invoked with a single initial root operator (with a name same as that of the prototype).
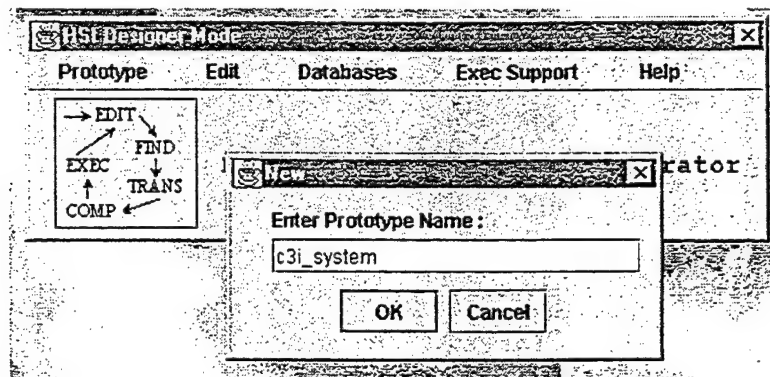
Figure 7. Creating a new prototype called C3I_System

APT allows the user to specify the requirements of prototypes as augmented dataflow graphs. Using the drawing tools provided by the PSDL editor, the user can create the top-level dataflow diagram of the c3i_system prototype as shown in Figure 8, where the c3i_system prototype is modeled by nine modules, communicating with each other via data streams. To model the dynamic behavior of these modules, the dataflow diagram is augmented with control and timing constraints. For example, the user may want to specify that the weapons_interface module has a maximum response time of 3 seconds to handle the event triggered by the arrival of new data in the weapon_status_data stream, and it only writes output to the weapon_emrep stream if the status of the weapon_status_data is damage, service_required, or out_of_ammunition. APT allow the user to specify these timing and control constraints using the pop-up operator property menu (Figure

9), resulting in a top-level PSDL program shown in Figure 10.
To complete the specification of the c3i_system prototype, the user must specify how each module will be implemented by choosing the implementation language for the module via the operator property menu. The implementation of a module can be in either the target programming language or PSDL. A module with an implementation in the target programming language is called an atomic operator. A module that is decomposed into a PSDL implementation is called a composite operator. Module decomposition can be done by selecting the corresponding operator in the tree-panel on the left side of the PSDL editor.

APT supports an incremental prototyping process. The user may choose to implement all nine modules as atomic operators (using dummy

158

components) in the first version, so as to check out the global effects of the timing and control constraints. Then, he/she may choose to decompose the comms_interface module into more detailed subsystems and implement the sub-modules with reusable components, while leaving the others as atomic operators in the second version of the prototype, and so on.
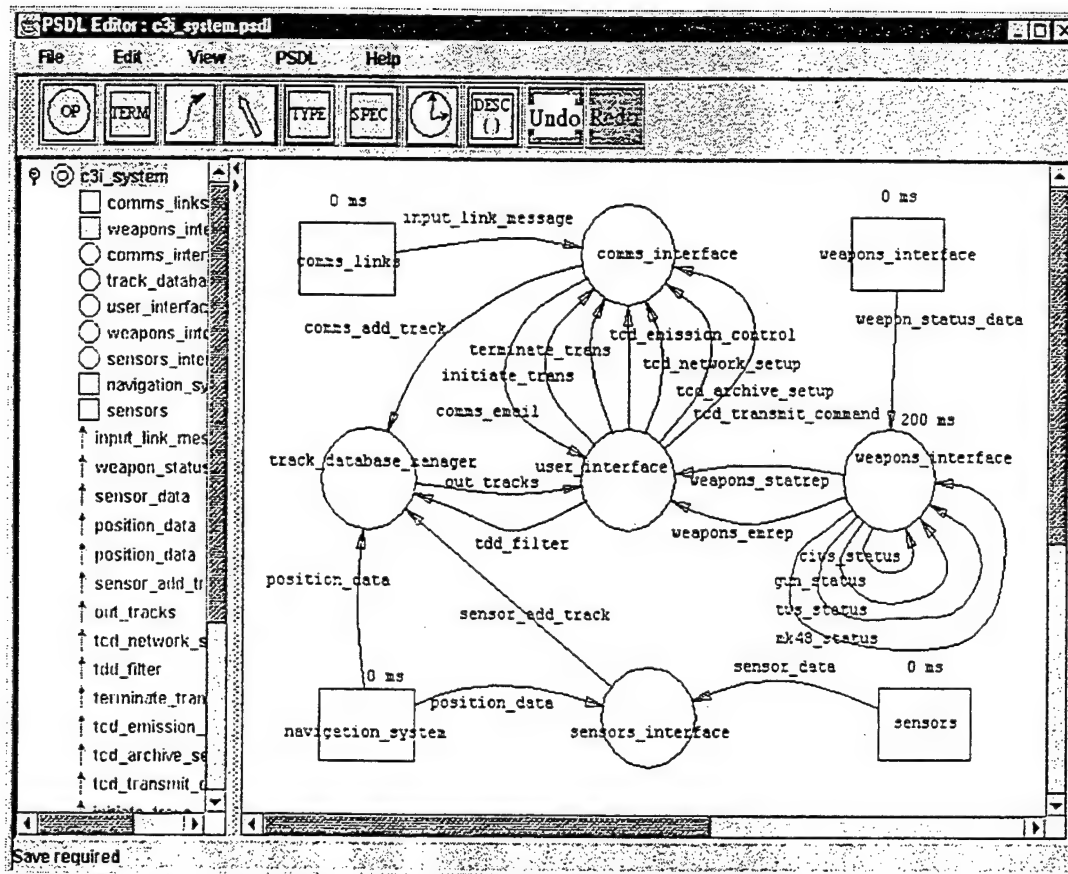
Figure 8. Top-level Dataflow Diagram of the c3i_system.
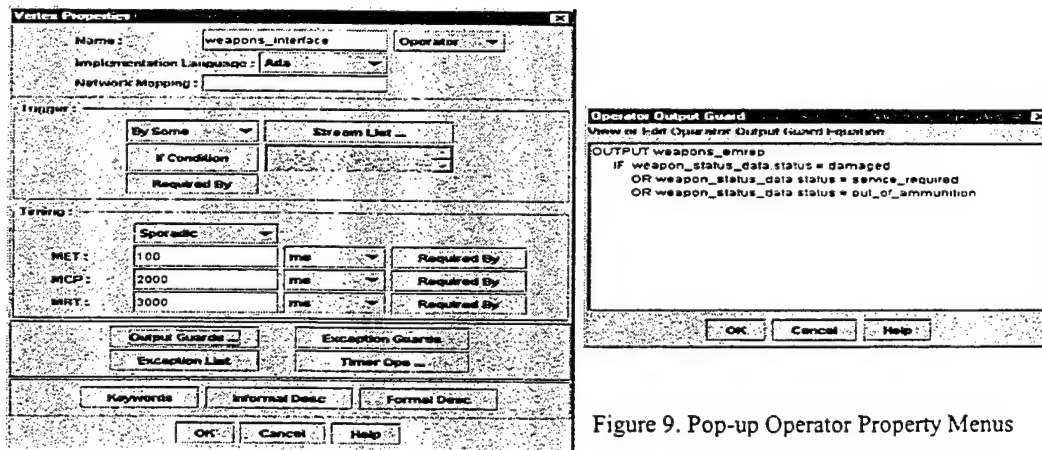
Figure 9. Pop-up Operator Property Menus

```
OPERATOR c3i_system
SPECIFICATION
    DESCRIPTION
        {This module implements a simplified version of
         a generic C3I workstation.}
END
IMPLEMENTATION
    GRAPH
```



```
    DATA STREAM
        -- Type declarations for the data streams in the graph go here.
    CONTROL CONTRAINTS
        OPERATOR comms_links                OPERATOR weapons_interface
            PERIOD 30000 MS                     TRIGGERED BY SOME
                                                    weapon_status_data
        OPERATOR navigation_system          MINIMUM CALLING PERIOD 2000 MS
            PERIOD 30000 MS                 MAXIMUM RESPONSE TIME 3000 MS
                                            OUTPUT
        OPERATOR sensors                        weapons_emrep
            PERIOD 30000 MS                     IF weapon_status_data.status =
                                                        damaged
        OPERATOR weapons_systems                OR weapon_status_data.status =
            PERIOD 30000 MS                             service_required
                                                OR weapon_status_data.status =
END                                                     out_of_ammunition
```

Figure 10. Top-level Specification of the c3i_system

To facilitate the testing of the prototypes, APT provides the user with an execution support system that consists of a translator, a scheduler and a compiler. Once the user finishes specifying the prototype, he/she can invoke the translator and the scheduler from the APT main interface to analyze the timing constraints for feasibility and to generate a supervisor module for each subsystem of the prototype in the target programming language. Each supervisor module consists of a set of driver procedures that realize all the control constraints, a high priority task (the static schedule) that executes the time-critical operators in a timely fashion, and a low priority dynamic schedule task that executes the non-time-critical operators when there is time available. The supervisor module also contains information that enables the compiler to incorporate all the software components required to implement the atomic operators and generate the binary code automatically. The translator/scheduler also generates the glue code needed for timely delivery of information between subsystems across the target network.

For prototypes which require sophisticated graphic user interfaces, the APT main interface provides an interface editor to interactively sculpt the interface. In the c3i_system prototype, we choose to decompose the comms_interface, the track_database_manager and the user_interface modules into subsystems, resulting in hierarchical design consisting of 8 composite operators and twenty-six atomic operators. The user interface of the prototype has a total of 14 panels, four of which are shown in Figure 11. The corresponding Ada program has a total of 10.5K lines of source code. Among the 10.5K lines of code, 3.5K lines comes from supervisor module that was generated automatically by the translator/scheduler and 1.7K lines that were automatically generated by the interface editor [9].

4 CONCLUSION

APT has been used successfully as a research tool in prototyping large warfighter control systems (e.g. the command-and-control station, cruise missile flight control system, missile defense systems) and demonstrated its capability to support the development of large complex embedded software. Specific payoffs include:

(1) Formulate/validate requirements via prototype demonstration and user feedback
(2) Assess feasibility of real-time system designs
(3) Enable early testing and integration of completed subsystems
(4) Support evolutionary system development, integration and testing
(5) Reduce maintenance costs through systematic code generation
(6) Produce high quality, reliable and flexible software
(7) Avoid schedule overruns

In order to evaluate the benefits derived from the practice of computer-aided prototyping within the software acquisition process, we conducted a case study in which we compared the cost (in dollar amounts) required to perform requirements analysis and feasibility study for the c3i system using the 2167A

process, in which the software is coded manually, and the rapid prototyping process, where part of the code is automatically generated via APT [3]. We found that, even under very conservative assumptions, using the APT method resulted in a cost reduction of $56,300, a 27% cost saving. Taking the results of this comparison, then projecting to a mission control software system, the command and control segment (CCS), we estimated that there would be a cost saving of 12 million dollars. Applying this concept to an engineering change to a typical component of the CCS software showed a further cost savings of $25,000.

Figure 11. User Interface of the c3i_system

5 REFERENCES

[1] L. Bernstein, "Forward: Importance of Software Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 9-14, 1996.

[2] I. Duranlioglu, *Implementation of a Portable PSDL Editor for the Heterogeneous Systems Integrator*, Master's thesis, Naval Postgraduate School, Monterey, California, March 1999.

[3] M. Ellis, *Computer-Aided Prototyping Systems (APT) within the software acquisition process: a case study*, Master's thesis, Naval Postgraduate School, Monterey, California, June 1993.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 1995.

[5] B. Kraemer, Luqi and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Transaction on Software Engineering*, 19(5), pp. 453-477, 1993.

[6] Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transaction on Software Engineering*, 14(10), pp. 1409-1423, 1988.

[7] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, 5(2), pp. 66-72, 1988.

[8] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer*, pp. 111-112, September 1991.

[9] Luqi, "Computer-Aided Prototyping for a Command-and-Control System Using APT", *IEEE Software*, 9(1), pp. 56-67, 1992.

[10] Luqi, "System Engineering and Computer-Aided Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 15-17, 1996.

[11] TAE Plus Programmer's Manual (Version 5.1). Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

Conceptual Level Graph Theoretic Design and Development of Complex Information System

S. Pramanik[2]　　　　S. Choudhury[1]　　　　N. Chaki[1]　　　　S.Bhattacharya[3]

Abstract: *This paper introduces a graph-oriented model for conceptual level design of large, complex information systems. This has been shown to be highly effective to the system designer from the perspectives of maintainability and upgradability. Basically due to the flat structure and the lack of holding multidimensional data, the relational model does not provide a structural approach to the system designer. The other alternative object oriented model offer the structured approach but also not able to describe various intermodular relationships spread over same or different levels within a data model. The graph data model also allows dynamic regrouping of related entities at the designers' level. We have proposed the appropriate data structure and the corresponding DDL has also been developed. Test runs on simulated environment further establishes its computational efficiency.*

Keywords: Graph oriented data model, Semantic view, Functional abstraction, Encapsulation of data and relationships.

[1]Department of Computer Science and Engineering, University of Calcutta, Calcutta :700 009, India,
E-mail: sankha@cucc.ernet.in
[2]Department of Computer Science, Michigan State University, East Lansing,MI, USA
[3]Department of Computer Science and Engineering, Jadavpur University, Calcutta :700 032, India
Communicating Author: S.Choudhury

Conceptual Level Graph Theoretic Design and Development of Complex Information System

Abstract: This paper introduces a graph-oriented model for conceptual level design of large, complex information systems. This has been shown to be highly effective to the system designer from the perspectives of maintainability and upgradability. Basically due to the flat structure and the lack of holding multidimensional data, the relational model does not provide a structural approach to the system designer. The other alternative object oriented model offer the structured approach but also not able to describe various intermodular relationships spread over same or different levels within a data model. The graph data model also allows dynamic regrouping of related entities at the designers' level. We have proposed the appropriate data structure and the corresponding DDL has also been developed. Test runs on simulated environment further establish its computational efficiency.

Keywords: Graph oriented data model, Semantic view, Functional abstraction, Encapsulation of data and relationships.

1. Introduction

The environments in which database management systems are being used have changed rapidly in the last several years. Although the relational model has made prominent contribution in the research of DBMS, recent database applications are outgrowing this model. The table based relational model is not the best approach to express complex and diverse databases. In this model, relationships among records are not structurally specified and due to this flat structure of the relational model, this is not useful to a user attempting to comprehend the logical structure actually existing in a schema. The alternative idea provide the concept of a class which can encapsulate homogeneous objects but there are no direct means to describe the mutual relationships amongst the objects within a class or to express the intermodular relationships spread over same or different levels. So our goal is to design a data model providing a structural approach which retains the desirable properties of the relational and object oriented model and simultaneously overcome the bottleneck of these schemes through the incorporation of some new features. In this effort, a graph based data model at conceptual level having the concept of functional abstraction has been developed. The significant improvement is expected corresponding tograph model in the context of maintainability, adaptability and transparency from the view of a system designer.

Here we discuss related work done in the areas of graph-based data models, object oriented approach in graph data models, semi-structured data and view update. Abiteboul in [2] uses semi-structured data that is neither raw data (file systems) nor strictly typed (table-oriented or object oriented). Even if semi-structured data may have a structure, this structure is often implicit, and not as rigid or regular as that found in standard database systems. In [3] an OQL like query language extended with information retrieval tools is proposed to query SGML and HTML documents. Buneman in [5] defines semi-structured data as that for which the information normally associated with a schema is

contained within the data itself. An attempt has been made to represent semi-structured data as graph like or tree like structure, where edges are labeled representing data types and leaves stand for raw data. In [6] a query language (UnQL) is adapted, which solves some of the limitations of SQL like languages for semi-structured data. Related work is also being conducted in both the area of semi-structured data access and querying. The graph data model presented here uses similar structure as in the ER data model [7], representing atomic entities by nodes and relations among them by links. However the database schema represented by the ER data model is not accessible by the DBMS, whereas in the graph data model the structure of the database is represented as part of the graph database itself. In [8] a graph model is proposed as underlying unified data model to access different databases expressed in standard data models. The query language is formally defined in terms of graphical primitives (atomic queries). A global information management system was developed providing a global framework where data on the web is accessed through conceptual views. GOOD [9,10,11] started as a database interface, then evolved as a graph object oriented database system. Actually, it is a graph representation of an object oriented database, where nodes represent objects and links represent relationships between objects. The GRAS data model [12] relies on attributed graphs. In this model, objects are represented by typed nodes, which may carry attributes. Relations between objects are modeled by bi-directional edges. In our model, we are trying to focus more on the concept of semantic groups providing the concept of functional abstraction for querying and updating data model but all the previous works are focused on defining a new approach to represent the graph data model itself.

Our goal is to provide a tool to the system designer level for describing and maintaining a complex semi-structured information system in a better way. So we have proposed a methodology to develop a directed graph model in the logical level as (V,E) where a node V represents a basic data object or a functionally abstracted module and an edge implies the binary relationships between the entities present in the graph data model. In the graph model we can encapsulate the nodes of lower level under a functional abstraction node from a specific semantic view. There is no restriction on the existence of relationships among the nodes in a graph. Based on the graph based data model framework, we have developed a data description language (DDL) for easy description and modification of the entity and their relationships within a complex information system. A quite user-friendly script is provided to the system designer for easy description of the conceptual level. In DDL, we have generated a friendly script for the system designer; a mathematical script has been generated also for each statement (e.g. relation, encapsulation) of the designer script and according to the operation described in the mathematical script the software will be executed generating the data structure as a output. These entire concepts have been crystallized in the form of a software tool, which has also been subsequently implemented.

2. The proposed data model and corresponding data structure

The conceptual level of a semi-structured information system is represented by the graph model depicted in fig.1. Here the basic instances of entity (lowest level vertices) or the functionally abstracted module is indicated by the vertex and the relationship among them by directed edges. In this graph data model the vertices indicated by triangle, square and circle indicate the node in the lowest level, intermediate level and the top most level respectively. The concept of encapsulation is implemented within the graph with respect to a functional abstraction node from a specific scientific view e.g. the nodes 4,5,6,7 and 8 are encapsulated in a same class under the functional abstraction node 2 reflecting a specific semantic view. The parallel edge between the nodes 4 and 5 indicate the existence of two different relation declared from two different semantic, declared with respect to abstraction node 1 and 2. It has been suggested a suitable data structure to declare the conceptual level of the data model

depicted in fig.1. A pointer array maintains the growth of this graph – with each element of the array representing a vertex in the graph. Each element of array points to a doubly linked list, right link maintains the set of vertices encapsulated by it and the left link points to the set of vertices within each of which it exists as a vertex of the encapsulated subgraph.
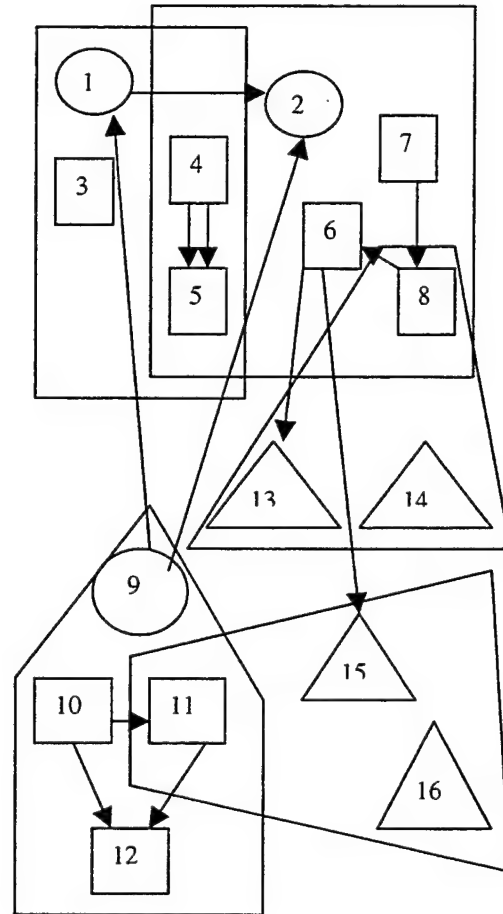
(Figure 1 : The Data Model)

Each element of the linked list is a structure of three elements –the vertex no., type of relation (i.e. encapsulation is represented by tag L and direct edge is denoted by tag E) and the functional abstraction node on which the relation is based on. The assumption has been made for creation of the data structure that the highest level vertices 1,2,3 are encapsulated within the vertex 0, which is treated as the top most level node. This is indicated by a high value in the left link of the vertex indicated by 0. The right link of node 1 indicates that an encapsulation class has been formed with the member 3,4 and 5 under the functional abstraction node 1 and there is a direct edge from 1 to 2 defined with respect to abstraction node 0. Similarly the left link indicates that the functional abstraction node 1 itself is encapsulated as a member under the node 0 and there is a direct edge from 9 to the node 1.Also from the linked lists corresponding the node 4, we can say that the node 4 is encapsulated as a member in the two encapsulation class generated under the node 1 as well as 2 and there is a parallel edge between 4 and 5 defined with respect to higher level node 1 and 2 respectively.

		100/E	0	1/E	2/E	9/E		
	9/L/0	0/E	1	3/E	4/E	5/E	2/L/0	
9/L/0	1/L/0	0/E	2	4/E	5/E	6/E	7/E	8/E
		1/E	3					
	2/E	1/E	4	5/L/1	5/L/2			
	2/E	1/E	5					
	8/L/2	2/E	6	13/L/2	15/L/2			
		2/E	7	8/L/2				
	7/L/2	2/E	8	13/E	14/E	6/L/2		
		0/E	9	10/E	11/E	12/E	1/L/0	2/L/0
		9/E	10	11/L/9	12/L/9			
		9/E	11	15/E	16/E	12/L/9		
11/L/9	10/L/9	9/E	12					
	6/L/2	8/E	13					
		8/E	14					
	6/L/2	11/E	15					
		11/E	16					

(Table 1: Data structure for figure 1 Data model)

3. Data Description Language

A data description language is also introduced in this paper by which the system designer will be able to easily describe the graph based data model and also can modify according to the need of the application. A user-friendly script is provided to designer for easy description of the data model. The equivalent mathematical script (relations) is being generated and executed through the procedures provided in the software producing the data structure i.e., the data model as an output.

3.1. Creation of the graph model

The DDL provided here have a two-fold job; one is to create the graph model depending upon the information available to system designer initially and modify it according to the requirement of the application. In this section we have described that how the data model depicted in fig.1 will be created as per this DDL.

We consider that there are three basic operations from the data description point of view.
- a. Creation of nodes, as an element of an array.
- b. Encapsulation of nodes belonging to a semantic class with respect to a higher level functional abstraction node.
- c. Declaration of direct relationship amongst nodes within the graph.

The syntax of the user-friendly script for the operations referred above as a to c is given below.

CREATE GRAPH [GRAPHNAME] [NO OF NODES];
ENCAP [CLASSNAME] [MEMBER OF CLASS] UNDER [FUNCTIONAL ABSTRACTION NODE];
CREATE REL [RELATION NAME] WITH [CLASSNAME] FOR [NODES INVOLVED IN RELATION]; So to declare the data model of fig.1 the designer have to declare the data and their relations according to the syntax already given.

CREATE GRAPH G1 17; * Initially we want to create a graph model named G1 of 17 nodes.
ENCAP E0 [1,2,9] UNDER 0; * The nodes 1,2 and 9 will be encapsulated under the node 0 and it will be identified by relation E0.
CREATE REL R1 WITH E0 FOR [1,2];
CREATE REL R2 WITH E0 FOR [9,1];
CREATE REL R3 WITH E0 FOR [9,2]; * This statement indicates that the relation named R1 reflects an edge between 1 and 2 defined with respect to the functional abstraction node mentioned in relation E0 i.e. 0. Instead of the above three statements, we can write CREATE REL R1,R2,R3 WITH E0 FOR [1,2],[9,1],[9,2].
ENCAP E1 [3,4,5] UNDER 1;
CREATE REL R4 WITH E1 FOR [4,5];
ENCAP E2 [4,5,6,7,8] UNDER 2;
CREATE REL R5 WITH E2 FOR [4,5];
CREATE REL R6 WITH E2 FOR [7,8];

In the above manner we have to express all the relations amongst nodes within the graph. Then the equivalent internal form will be generated after compilation and the corresponding script is described here.

ψ G1 [0–16]; * The symbols ψ and ϕ are used for creation and encapsulation respectively.

E0 = ϕ [1,2,9]^0;

R1 =   [1,2]/ E0; * Direct relation between 1 & 2 is defined with respect to the functional abstraction node present in relation E0.

R2 =   [9,1]/ E0;

R3 =   [9,2]/ E0;

E1 = ϕ [3,4,5]^1;

R4 =   [4,5]/E1;

E2 = ϕ [4,5,6,7,8]^2;

R5 =   [4,5]/E2;

R6 = [7,8]/E2;

Now these mathematical expressions, as declared by the designer, are treated as an input of the software (also provided in DDL) and the corresponding data structure is generated as output.

The complexity of the algorithm for development of data structure from a graph of n vertices, as specified by the system designer, is $O(n^2)$. This has been tested in a simulated environment. A random graph with degree varying from 4 to a maximum number of 10 has been considered as input to the algorithm and the corresponding data structure has been generated. The execution time has been plotted against the number of vertices as shown below.

Number of nodes	Execution time (in secs)		
	Degree 4	Degree 7	Degree 10
200			0.054989
300		0.054989	0.054989
400		0.054989	0.10989
500	0.054989	0.10989	0.10989
600	0.10989	0.164835	0.21978
700	0.10989	0.274725	0.384615
800	0.164835	0.32967	0.549451
900	0.21978	0.43956	0.659341

Figure showing Execution time (in Secs) on the y-axis (0 to 0.8) versus Number of nodes (in 100s) on the x-axis (0 to 10). Legend: D4, D7, D10, Poly. (D10), Poly. (D7), Poly. (D4).
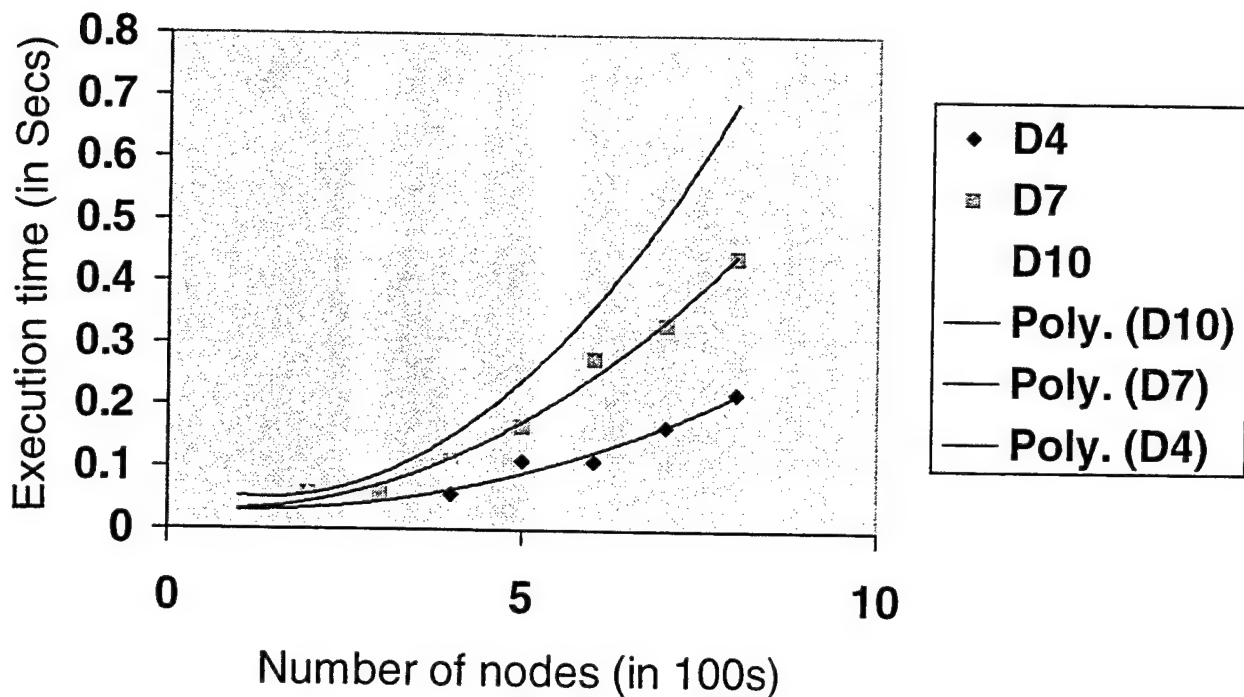
3.2. Modification of the graph model

In context to the modification of a graph model the basic operations are:

   a. Insertion of node(s) in the graph model.
   b. Deletion of an existing node.
   c. Modification of an encapsulation class from a different semantic view.
   d. Modification of an already existing edge.

The syntax of the user-friendly script of the operations referred here as a to d is given below.
OPEN GRAPH [GRAPH MODEL NAME]; * Initially the designer have to open the graph model G1 for modification.
INSERT NODE [NO. OF NODES];
DELETE NODE [NODES];

Generally the designer can delete only the lowest level nodes. The nodes selected for deletion including the relations involving these nodes will be deleted as a result of this operation. But if this attempt has been made for any functional abstraction node, all the nodes encapsulated within it will also be deleted including the specified node after getting an assurance for this operation from the system designer.

The operation referred above as c can be implemented by insertion of a node into an encapsulation class from another class or by substitution of a group of nodes by the members of a different class. In this case, all the relations involving the nodes take place in the operation are deleted. Again the designer have to define the new relationship from a different semantic aspect.
INSERT NODE(S) [NODES] OF CLASS [CLASSNAME] WITHIN CLASS [CLASSNAME];
MODIFY NODE(S) [NODES] OF CLASS [CLASSNAME] BY [NODES] OF CLASS [CLASSNAME];

There are two cases regarding the operation d, either we want to delete an existing relation or to insert a new relation between a pair of node with respect to an abstraction node.
DELETE REL [NODE1,NODE2,ABSTRACTION CLASS];

INSERT REL [NODE1,NODE2,ABSTRACTION CLASS];
    The specified graph model must be closed after the completion of the modification.
CLOSE GRAPH [GRAPH MODEL NAME].

    Let us describe the modification of the graph model with an example. We assume the system designer want to perform the following modifications on the graph model of fig.1.

1. Insert two new nodes in the graph model.
2. Enter these two nodes in the encapsulation class headed by functional abstraction node 2.
3. Replace the node 3 of encapsulation class headed by node 1 by the node 7 of the class under node 2.
4. Create an edge from node 7 to 5 with respect to node 1.
5. Delete the edge between 4 and 5 defined with respect to node 2.

    To perform the modifications mentioned above, the designer script will be:
OPEN GRAPH G1;
INSERT NODES [2];
INSERT NODES [17,18] OF CLASS E0 WITHIN CLASS E2;
MODIFY NODE [3] OF CLASS E1 BY [7] OF CLASS E2;
INSERT REL [7,5,E1];
DELETE REL [7,8,E2];
CLOSE GRAPH G1.
    The equivalent mathematical script involving the new modified relations will be:
OPEN G1;
ψ [17-18]; * Create two new nodes in the graph model G1.
$E0 = \phi[1,2,9,17,18]^0$; * By default , these two nodes are encapsulated within E0.
$E2 = \phi[4,5,6,7,8,17,18]^2$; *The modified class relation E2.
$E0 = \phi[1,2,9]^0$;
$E1 = \phi[4,5,7]^1$;
$E2 = \phi[3,4,5,6,17,18]^2$;
DEL REL R6; * Delete the previous relation involving node 7,named R6, as the previous relation may not exist from the new semantic view. If required, re-describe the relation.
$R6 = [7,5]/E1$; * A new relation named R6 is generated.
DEL REL R5; * The relation between 7 & 8 with respect to node 2 (named R5) will be deleted.
CLOSE G1.
    The mathematical script written above for the modification is also executed through the software and the data structure of the conceptual data model will be modified accordingly.

4. Graph Data Model in Distributed Computing Environment

    In recent years, almost all of the software should be compatible to distributed environment due to the increasing trend towards the distribution of computer systems over multiple sites that are interconnected via a communication network. So the distributed database concept with respect to our data model implies that the graph designed by system designer must be spread over the sites of a computer network. The fragmentation amount of the graph totally depends on the nature of specific application e.g. What type of queries will be processed at a specific site; What are the necessary information related to these queries, etc. Still in this section we propose some general fragmentation methodologies of the graph model to achieve the improved performance.Depending upon the nature of the queries and the related necessary information to process these queries, the relevant portion of the graph, i.e. the semantic groups must be distributed amongst the sites. We termed this method as Graph fragmentation. It may be the

case that one can keep the necessary information (occurrences) within a group instead of storing all members of that group. This is decided dynamically through generation of certain constraints. In that case the entire copy of the semantic group must be kept in another site to avoid the loss of information.

In some cases, we have to keep more than one copy of the same semantic group in different sites. In spite of the chances of generating inconsistency during updation, the replication is to be allowed to increase the availability and to reduce the communication cost for accessing data from different sites. In our data model, described in section 2, we have allowed to declare the relationship between the meber of different groups spread over different levels. After fragmentation, the mutual relationship within the group must belong to the subgraph present in the local site. A table, named Link table, has been maintained to keep track of information regarding relationship of any node in the local site with another node in some other site. This has been illustrated with an example in the next paragraph.

Suppose there are three sites S1, S2 and S3 with respect to abstract data model as depicted in fig. 1 and analysing the query to be processed, the designer has taken the decision about distribution in the following manner.

Site 1:   Semantic groups encapsulated under functional abstraction node 1 and 2.
Site 2:   Semantic groups encapsulated under functional abstraction node 2 and 8.
Site 3:   Semantic groups encapsulated under functional abstraction node 9 and 11.

According to the fragmentation scheme described above, the data structure depicted in Table 1 will also be decomposed amongst the sites. The figure 2 implies the subgraph belongs to site 1 and the Table 2 indicates the corresponding data structure. Here the dotted lines and vertices indicate that these are not belonging within the site 1 but for query processing these vertices or links may be required.
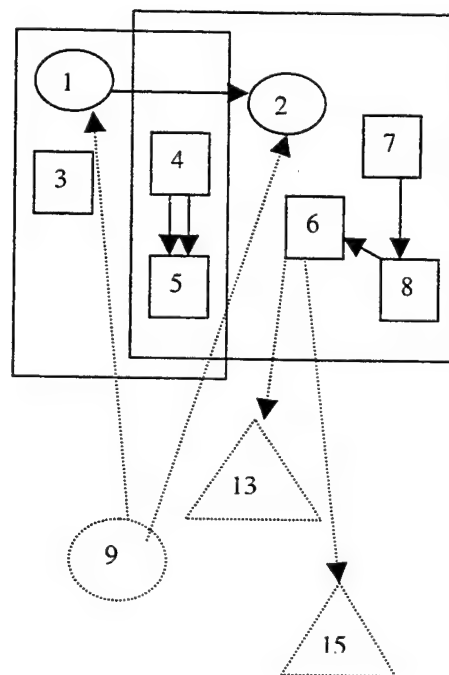
(Figure 2 : Fragmented Graph model for site 1)

		100/E	0	1/E	2/E			
	9/L/0	0/E	1	3/E	4/E	5/E	2/L/0	
9/L/0	1/L/0	0/E	2	4/E	5/E	6/E	7/E	8/E
		1/E	3					
	2/E	1/E	4	5/L/1	5/L/2			
	2/E	1/E	5					
	8/L/2	2/E	6	13/L/2	15/L/2			
		2/E	7	8/L/2				
	7/L/2	2/E	8	6/L/2				

(Table 2: Data structure for Site 1)

Now if the queries to be processed in site 1 are generally based on retrieval, then the nodes 13, 15 and 9 should be replicated into site 1. Otherwise, a link table for site 1 is maintained using which information can be fetched from different sites to process the queries.

Site Number	Vertex 1	Vertex 2
2	6	13
3	6	15
3	9	1
3	9	2

(Table 3 : Link Table for site 1)

The task for creating site 1 can be accomplished by the following set of commands provided in the proposed data model.

Open graph G1;
Fragment group 1,2 into site1;

In a similar way, the other sites can be created and be joined back to regenerate the data model in figure 1 by the graph join operation provided in our data model.

5. Conclusion

In this paper, an attempt has been made to present an alternative approach for storage and maintenance of semi-structured data based information system. A better performance may be obtained from our data model due to the point mentioned below.

Maintainability: The relational model is not providing a structured approach of the entities present in a large information system. So to find out the actual relations among entities scattered through different tables or the relations between tables, the designer has to derive the relations via common attributes searching through the tables. In a complicated large system, it will be a cumbersome process. But due to the provision of the structured approach in our proposed model, the designer can easily find out the relationships between some attributes (lowest level node) or some functional abstraction node directly via the option provided in our DDL.

Adaptability: It will be an ideal condition to the system designer, if all information regarding the application is clearly known at the right of the beginning. Unfortunately, in practical, we have to initiate

the design process with only limited knowledge and the system is going to be gradually enriched with the inclusion of new information. So the data model should provide the feature of easy inclusion of new information on the existing data model. The proposed model is flexible one to offer the designer to easily incorporate new information as node in the graph and to describe the relations of the new nodes with the existing nodes through edges. It also provides the facility to redefine the relationships from a different semantic view and accordingly the designer can also maintain the different semantic view of the same data model as per the requirement. So this model is a really adaptable one providing more than one view of same data model with respect to separate semantic to the system designer level. (This view is totally different from the view provided to the end user level)

Context sensitivity of the relations: The concept of functional abstraction is introduced to increase the effectiveness of the model. The designer will be able to formulate the behavioral aspects of the entities by forming an encapsulation class with respect to a functional abstraction node and can declare the mutual relationships among the members of the class. All these relations are context sensitive i.e. declared from a specific semantic which is incorporated within functional abstraction node e.g. the parallel edge within node 4 and 5 in fig.1 is context sensitive; one is defined with respect to abstraction node 1 and the other with respect node 2. So the significant improvement has been expected for this graph based data model in the context of the points mentioned in this section. The present work may further be consolidated by treating a Table as node in lieu of occurances of the attributes to maintain the user-friendlyness at the end user level.

References :

[1] S.Choudhury, N Assem, S.Pramanik and S.Bhattacharya, Graph theoretic modelling of semi-structured Information system based on functional abstraction, *Proc. of IASTED International Conference on Applied Modelling and Simulation*, 1998, 518-522.

[2] S Abiteboul and A Bonner, Objects and views, Proceedings of the 1991 ACMSIGMOD International Conference on Management of data In SIGMOD Record Vol.20, No.2, May 1991, 238-247.

[3] S Abiteboul, S Cluet, V Christophides, T Milo, G Moerkotte and J Simeon, Querying documents in object databases, *International Journal on Digital Libraries*, vol.1, April 1997.

[4] S Abiteboul and V Vainu, Queries and computation on the web, *Proc. of International Conference on Database Theory*, 1997.

[5] P Buneman, Semistructed data, *Proc. of ACM PODS'97*, 1997, 117-121.

[6] P Buneman, S Davidson, G Hillbrand, D Sucin, A query language and optimization techniques for unstructured data, *ACM SIGMOD*, NO.6, 1996, 505-516.

[7] P Chen, The Entity Relationship mode-Toward a unified view of data, *ACM transactions on Database System*, vol.1, 1976, 9-36.

[8] M Consens and A Mendelzon, Hy+: A hypergraph based query and visualization on system, *ACM SIGMOD Record*, vol.22, June 1993, 511-516.

[9] M Gyssens, J Paradaens, J V Bussche and D Gucht, A Graph oriented object database model, *IEEE Transactions on Knowledge and Engineering*, vol.6, Aug 1994, 572-586.

[10] M Gemis, J Paradaens, I Thyssens and J Bussche, GOOD : a graph oriented object database system, *ACM SIGMOD Record*, vol.22, June 1993, 505-510.

[11] M Gyssens, J Paradaens and D Gucht, A Graph oriented object model for database end user interfaces, *ACM SIGMOD Record*, vol.19: 2, 1990, 24-33.

[12] N Kiesel, A Schueer and B Westfechtel, GRAS, a graph oriented (software) engineering database system, *Information systems*, vol. 20, no.1, 1995, 21-51.

Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics

Norman F. Schneidewind, *Fellow, IEEE*

Abstract—In analyzing the stability of a maintenance process, it is important that it not be treated in isolation from the reliability and risk of deploying the software that result from applying the process. Furthermore, we need to consider the efficiency of the test effort that is a part of the process and a determinate of reliability and risk of deployment. The relationship between product quality and process capability and maturity has been recognized as a major issue in software engineering based on the premise that improvements in process will lead to higher quality products. To this end, we have been investigating an important facet of process capability—stability—as defined and evaluated by trend, change, and shape metrics, across releases and within a release. Our integration of product and process measurement serves the dual purpose of using metrics to assess and predict reliability and risk and to evaluate process stability. We use the NASA Space Shuttle flight software to illustrate our approach.

Index Terms—Maintenance process stability, product and process integration, reliability risk.

---

1 INTRODUCTION

MEASURING and evaluating the stability of maintenance processes is important because of the recognized relationship between process quality and product quality [7]. We focus on the important quality factor *reliability*. A maintenance process can quickly become unstable because the very act of installing software changes the environment: pressures operate to modify the environment, the problem, and the technological solutions. Changes generated by users and the environment and the consequent need for adapting the software to the changes is unpredictable and cannot be accommodated without iteration. Programs must be adaptable to change and the resultant change process must be planned and controlled. According to Lehman, large programs are never completed, they just continue to evolve [11]. In other words, with software, we are dealing with a moving target. Maintenance is performed continuously and the stability of the maintenance process has an effect on product reliability. Therefore, when we analyzed the stability of the NASA Space Shuttle software maintenance process, it was important to consider the reliability of the software that the process produces. Furthermore, we needed to consider the efficiency of the test effort that is a part of the process and a determinate of reliability. Therefore, we integrated these factors into a unified model, which allowed us to measure the influence of maintenance actions and test effort on the reliability of the software. Our hypothesis was that these metrics would exhibit trends and other characteristics over time that would be indicative of

the stability of the process. Our results indicate that this is the case.

We conducted research on the NASA Space Shuttle flight software to investigate a hypothesis of measuring and evaluating maintenance stability. We used several metrics and applied them across releases of the software and within releases. The trends and shapes of metric functions over time provide evidence of whether the software maintenance process is stable. We view stability as the condition of a process that results in increasing reliability, decreasing risk of deployment, and increasing test effectiveness. In addition, our focus is on process stability, not code stability. We explain our criteria for stability; describe metrics, trends, and shapes for judging stability; document the data that was collected; and show how to apply our approach. Building on our previous work of defining maintenance stability criteria and developing and applying trend metrics for stability evaluation [15], in this paper we review related research projects, introduce shape metrics for stability evaluation, apply our change metric for multiple release stability evaluation, consider the functionality of the software product in stability evaluation, and interpret the metric results in terms of process improvements.

Our emphasis in this paper is to propose a unified product and process measurement model for product evaluation and process stability analysis. The reader should focus on the model principles and not on the results obtained for the Shuttle. These are used only to illustrate the model concepts. In general, different numerical results would be obtained for other applications that use this model.

Section 2 reviews related research. In Section 3, the concept of stability is explained and trend and shape metrics are defined. Section 4 defines the data and the NASA Space Shuttle application environment. Section 5 gives an analysis of relationships among maintenance, reliability, test effort, and risk, while Section 6 discusses

---

- *The author is with the Computer and Information Sciences and Operations Division, Naval Postgraduate School, Monterey, CA 93943.*
  E-mail: nschneid@nps.navy.mil.

both long term (i.e., across releases) and short term (i.e., within a release), as applied to the NASA Space Shuttle. Section 7 discusses our attempt to relate product metrics to process improvements and to the functionality and complexity of the software. Conclusions are drawn in Section 8.

2 RELATED RESEARCH AND PROJECTS

A number of useful related maintenance measurement and process projects have been reported in the literature. Briand et al. developed a process to characterize software maintenance projects [3]. They present a qualitative and inductive methodology for performing objective project characterizations to identify maintenance problems and needs. This methodology aids in determining causal links between maintenance problems and flaws in the maintenance organization and process. Although the authors' have related ineffective maintenance practices to organizational and process problems, they have not made a linkage to product reliability and process stability.

Gefen and Schneberger developed the hypothesis that maintenance proceeds in three distinct serial phases: corrective modification, similar to testing; improvement in function within the original specifications; and the addition of new applications that go beyond the original specifications [5]. Their results from a single large information system, which they studied in great depth, suggested that software maintenance is a multiperiod process. In the NASA Space Shuttle maintenance process, in contrast, all three types of maintenance activities are performed concurrently and are accompanied by continuous testing.

Henry et al. found a strong correlation between errors corrected per module and the impact of the software upgrade [6]. This information can be used to rank modules by their upgrade impact during code inspection in order to find and correct these errors before the software enters the expensive test phase. The authors treat the impact of change but do not relate this impact to process stability.

Khoshgoftarr et al. used discriminant analysis in each iteration of their project to predict fault prone modules in the next iteration [10]. This approach provided an advance indication of reliability and the risk of implementing the next iteration. This study deals with product reliability but does not address the issue of process stability.

Pearse and Oman applied a maintenance metrics index to measure the maintainability of C source code before and after maintenance activities [13]. This technique allowed the project engineers to track the "health" of the code as it was being maintained. Maintainability was assessed but not in terms of process stability.

Pigoski and Nelson collected and analyzed metrics on size, trouble reports, change proposals, staffing, and trouble report and change proposal completion times [14]. A major benefit of this project was the use of trends to identify the relationship between the productivity of the maintenance organization and staffing levels. Although productivity was addressed, product reliability and process stability were not considered.

Sneed reengineered a client maintenance process to conform to the ANSI/IEEE Standard 1291, Standard for Software Maintenance [19]. This project is a good example of how a standard can provide a basic framework for a

process and can be tailored to the characteristics of the project environment. Although applying a standard is an appropriate element of a good process, product reliability and process stability were not addressed.

Stark collected and analyzed metrics in the categories of customer satisfaction, cost, and schedule with the objective of focusing management's attention on improvement areas and tracking improvements over time [20]. This approach aided management in deciding whether to include changes in the current release, with possible schedule slippage, or include the changes in the next release. However, the authors did not relate these metrics to process stability.

Although there were similarities between these projects and our research, our work differed in that we integrated: 1) maintenance actions, 2) reliability, 3) test effort, and 4) risk to the safety of mission and crew of deploying the software after maintenance actions, for the purpose of analyzing and evaluating the stability of the maintenance process.

3 CONCEPT OF STABILITY

3.1 Trend Metrics

To gain insight into the interaction of the maintenance process with product metrics like reliability, two types of metrics were analyzed: trend and shape. Both types are used to assess and predict maintenance process stability across (long term) and within (short term) releases after the software is released and maintained. Shape metrics are described in Section 3.2. By chronologically ordering metric values by release date, we obtain discrete functions in time that can be analyzed for trends across releases. Similarly, by observing the sequence of metric values as continuous functions of increasing test time, we can analyze trends within releases. These metrics are defined as empirical and predicted functions that are assigned values based on release date (long term) or test time (short term). When analyzing trends, we note whether an increasing or decreasing trend is favorable [15]. For example, an *increasing* trend in Time to Next Failure and a *decreasing* trend in Failures per KLOC would be favorable. Conversely, a *decreasing* trend in Time to Next Failure and an *increasing* trend in Failures per KLOC would be unfavorable. A favorable trend is indicative of maintenance stability if the functionality of the software has increased with time across releases and within releases. Increasing functionality is the norm in software projects due to the enhancement that users demand over time. We impose this condition because if favorable trends are observed, they could be the result of decreasing functionality rather than having achieved maintenance stability. When trends in these metrics over time are favorable (e.g., increasing reliability), we conclude that the maintenance process is *stable* with respect to the software metric (reliability). Conversely, when the trends are unfavorable (e.g., decreasing reliability), we conclude that process is *unstable*. Our research investigated whether there were relationships among the following factors: 1) maintenance actions, 2) reliability, and 3) test effort. We use the following types of trend metrics:

1. *Maintenance actions:* KLOC Change to the Code (i.e., amount of code changed necessary to add given functionality);
2. *Reliability:* Various reliability metrics (e.g., MTTF, Total Failures, Remaining Failures, and Time to Next Failure); and
3. *Test effort:* Total Test Time.

3.1.1 Change Metric

Although looking for a trend metric on a graph is useful, it is not a precise way of measuring stability, particularly if the graph has peaks and valleys and the measurements are made at discrete points in time. Therefore, we developed a Change Metric (CM), which is computed as follows:

1. Note the change in a metric from one release to the next (i.e., release j to release $j+1$).
2. If the change is in the desirable direction (e.g., Failures/KLOC decrease), treat the change in 1 as positive. If the change is in the undesirable direction (e.g., Failures/KLOC increase), treat the change in 1 as negative.
3. If the change in 1 is an increase, divide it by the value of the metric in release $j+1$. If the change in 1 is a decrease, divide it by the value of the metric in release j.
4. Compute the average of the values obtained in 3, taking into account sign. This is the change metric (CM). The CM is a quantity in the range $-1, 1$. A positive value indicates stability; a negative value indicates instability. The numeric value of CM indicates the degree of stability or instability. For example, 0.1 would indicate marginal stability and 0.9 would indicate high stability. Similarly, -0.1 would indicate marginal instability and -0.9 would indicate high instability. The standard deviation of these values can also be computed. Note that CM only pertains to stability or instability *with respect to the particular metric that has been evaluated* (e.g., Failures/KLOC). The evaluation of stability should be made with respect to a set of metrics and not a single metric. The average of the CM for a set of metrics can be computed to obtain an overall metric of stability.

3.2 Shape Metrics

In addition to trends in metrics, the shapes of metric functions provide indicators of maintenance stability. We use shape metrics to analyze the stability of an individual release and the trend of these metrics across releases to analyze long-term stability. The rationale of these metrics is that it is better to reach important points in the growth of product reliability sooner than later. If we reach these points late in testing, it is indicative of a process that is late in achieving stability. We use the following types of shape metrics:

1. Direction and magnitude of the slope of a metric function (e.g., failure rate decreases asymptotically with total test time). Using failure rate as an example within a release, it is desirable that it rapidly

decrease toward zero with increasing total test time and that it have small values.
2. Percent of total test time at which a metric function changes from unstable (e.g., increasing failure rate) to stable (e.g., decreasing failure rate) and remains stable. Across releases, it is desirable that the total test time at which a metric function becomes stable gets progressively smaller.
3. Percent of total test time at which a metric function increases at a maximum rate in a favorable direction (e.g., failure rate has maximum negative rate of change). Using failure rate as an example, it is desirable for it to achieve maximum rate of decrease as soon as possible, as a function of total test time.
4. Test time at which a metric function reaches its maximum value (e.g., test time at which failure rate reaches its maximum value). Using failure rate as an example, it is desirable for it to reach its maximum value (i.e., transition from unstable to stable) as soon as possible, as a function of total test time.
5. *Risk:* Probability of *not* meeting reliability and safety goals (e.g., time to next failure should exceed mission duration), using various shape metrics as indicators of risk. Risk would be low if the conditions in 1-4 above obtain.

3.3 Metrics for Long-Term Analysis

We use certain metrics only for long-term analysis. As an example, we compute the following trend metrics over a sequence of releases:

1. Mean Time to Failure (MTTF).
2. Total Failures normalized by KLOC Change to the Code.
3. Total Test Time normalized by KLOC Change to the Code.
4. Remaining Failures normalized by KLOC Change to the Code.
5. Time to Next Failure.

3.4 Metrics for Long- and Short-Term Analysis

We use other metrics for both long-term and short-term analysis. As an example, we compute the following trend (1) and shape (2, 3, 4, and 5) metrics over a sequence of releases and within a given release:

1. Percent of Total Test Time required for Remaining Failures to reach a specified value.
2. Degree to which Failure Rate asymptotically approaches zero with increasing Total Test Time.
3. Percent of Total Test Time required for Failure Rate to become stable and remain stable.
4. Percent of Total Test Time required for Failure Rate to reach maximum decreasing rate of change (i.e., slope of the failure rate curve).
5. Maximum Failure Rate and Total Test Time where Failure Rate is maximum.

4 DATA AND EXAMPLE APPLICATION

We use the NASA Space Shuttle application to illustrate the concepts. This large maintenance project has been evolving

TABLE 1
Characteristics of Maintained Software Across *NASA Space Shuttle* Releases (Part 1)

Operational Increment	Release Date	Launch Date	Mission Duration (Days)	Reliability Prediction Date	Total Post Delivery Failures	Failure Severity
A	9/1/83	No Flights		12/9/85	6	One 2 Five 3
B	12/12/83	8/30/84	6	8/14/84	10	Two 2 Eight 3
C	6/8/84	4/12/85	7	1/17/85	10	Two 2 Seven 3 One 4
D	10/5/84	11/26/85	7	10/22/85	12	Five 2 Seven 3
E	2/15/85	1/12/86	6	5/11/89	5	One 2 Four 3
F	12/17/85				2	Two 3
G	6/5/87				3	One 1 Two 3
H	10/13/88				3	Two 1 One 3
I	6/29/89				3	Three 3
J	6/18/90	8/2/91	9	7/19/91	7	Seven 3
K	5/2/91				1	One 1
L	6/15/92				3	One 1 One 2 One 3
M	7/15/93				1	One 3
N	7/13/94				1	One 3
O	10/18/95	11/19/96	18	9/26/96	5	One 2 Four 3
P	7/16/96				3	One 2 Two 3
Q	3/5/97				1	One 3

with increasing functionality since 1983 [2]. We use data collected from the developer of the flight software of the NASA Space Shuttle, as shown in Table 1, Part 1, and Table 2, Part 2. These tables show Operational Increments (OIs) of the NASA Space Shuttle: OIA... OIQ, covering the period 1983-1997. We define an OI as follows: a software system comprised of modules and configured from a series of builds to meet NASA Space Shuttle mission functional requirements [16]. In Part 1, for each of the OIs, we show the Release Date (the date of release by the contractor to NASA), Total Post Delivery Failures, and Failure Severity (decreasing in severity from "1" to "4"). In Part 2, we show the maintenance change to the code in KLOC (source language changes and additions) and the total test time of the OI. In addition, for those OIs with at least two failures, we show the computation of MTTF, Failures/KLOC, and Total Test Time/KLOC. KLOC is an indicator of maintenance actions, not functionality [8]. Increased functionality, as measured by the increase in the size of principal functions loaded into mass memory, has averaged about 2 percent over the last 10 OIs. Therefore, if a stable process were observed, it could not be attributed to decreasing functionality. Also to be noted is that the software developer is a CMM Level 5 organization that has continually improved its process.

Because the flight software is run continuously, around the clock, in simulation, test, or flight, Total Test Time refers to continuous execution time from the time of release. For

TABLE 2
Characteristics of Maintained Software Across *NASA Space Shuttle* Releases (Part 2)

Operational Increment	KLOC Change	Total Test Time (Days)	MTTF (Days)	Total Failures/KLOC Change	Total Test Time/ KLOC Change (Days)
A	8.0	1078	179.7	0.750	134.8
B	11.4	4096	409.6	0.877	359.3
C	5.9	4060	406.0	1.695	688.1
D	12.2	2307	192.3	0.984	189.1
E	8.8	1873	374.6	0.568	212.8
F	6.6	412	206.0	0.303	62.4
G	6.3	3077	1025.7	0.476	488.4
H	7.0	540	180.0	0.429	77.1
I	12.1	2632	877.3	0.248	217.5
J	29.4	515	73.6	0.238	17.5
K	21.3	182			8.5
L	34.4	1337	445.7	0.087	38.9
M	24.0	386			16.1
N	10.4	121			11.6
O	15.3	344	68.8	0.327	22.5
P	7.3	272	90.7	0.411	37.3
Q	11.0	75			6.8

OIs where there was a sufficient sample size (i.e., Total Post Delivery Failures)—OIA, OIB, OIC, OID, OIE, OIJ, and OIO—we predicted software reliability. For these OIs, we show Launch Date, Mission Duration, and Reliability Prediction date (i.e., the date when we made a prediction). Fortunately, for the safety of the crew and mission, there have been few postdelivery failures. Unfortunately, from the standpoint of prediction, there is a sparse set of observed failures from which to estimate reliability model parameters, particularly for recent OIs. Nevertheless, we predict reliability prior to launch date for OIs with as few as five failures spanning many months of maintenance and testing. In the case of OIE, we predict reliability after launch because no failures had occurred prior to launch to use in the prediction model. Because of the scarcity of failure data, we made predictions using all severity levels of failure data. This turns out to be beneficial when making reliability risk

assessments using number of Remaining Failures. For example, rather than specifying that the number of predicted Remaining Failures must not exceed one severity "1," the criterion could specify that the prediction not exceed one failure of *any type*—a more conservative criterion [16].

As would be expected, the number of predelivery failures is much greater than the number of postdelivery failures because the software is not as mature from a reliability standpoint. Thus, a way around the insufficient sample size of recent OIs for reliability prediction is to use predelivery failures for model fit and then use the fitted model to predict postdelivery failures. However, we are not sure that this approach is appropriate because the multiple builds in which failures can occur and the test strategies used to attempt to crash various pieces of code during the predelivery process contrast sharply with the postdelivery

environment of testing an integrated OI with operational scenarios. Nevertheless, we are experimenting with this approach in order to evaluate the prediction accuracy. The results will be reported in a future paper.

5 RELATIONSHIP BETWEEN MAINTENANCE, RELIABILITY, RISK, AND TEST EFFORT

5.1 Metrics for Long-Term Analysis

We want our maintenance effort to result in increasing reliability of software over a sequence of releases. A graph of this relationship over calendar time and the accompanying CM calculations indicate whether the long-term maintenance effort has been successful as it relates to reliability. In order to measure whether this is the case, we use both predicted and actual values of metrics. We predict reliability in advance of deploying the software. If the predictions are favorable, we have confidence that the risk is acceptable to deploy the software. If the predictions are unfavorable, we may decide to delay deployment and perform additional inspection and testing. Another reason for making predictions is to assess whether the maintenance process is effective in improving reliability and to do it sufficiently early during maintenance to improve the maintenance process. In addition to making predictions, we collected and analyzed historical reliability data. These data show in retrospect whether maintenance actions were successful in increasing reliability. In addition, the test effort should not be disproportionate to the amount of code that is changed and to the reliability that is achieved as a result of maintenance actions.

5.1.1 Mean Time to Failure

We want Mean Time to Failure (MTTF), as computed by (1), to show an increasing trend across releases, indicating increasing reliability.

$$
\begin{aligned}
\text{Mean Time to Failure} = \ &\text{Total Test Time/Total} \\
&\text{Number of Failures During} \quad (1) \\
&\text{Test}
\end{aligned}
$$
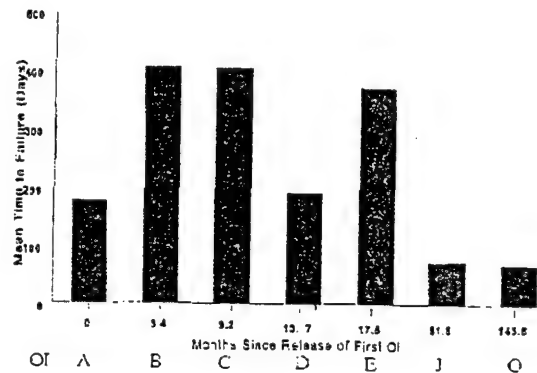
Fig. 1. Mean time to failure across releases.

5.1.2 Total Failures

Similarly, we want Total Failures (and faults), normalized by KLOC Change in Code, as computed by (2), to show a **decreasing** trend across releases, indicating that reliability is **increasing** with respect to code changes.

$$
\begin{aligned}
\text{Total Failures/KLOC} = \ &\text{Total Number of Failures} \\
&\text{During Test/KLOC Change} \quad (2) \\
&\text{in Code on the OI}
\end{aligned}
$$

We plot (1) and (2) in Fig. 1 and Fig. 2, respectively, against Release Time of OI. This is the number of months since the release of the OI, using "0" as the release time of OIA. We identify the OIs at the bottom of the plots. Both of these plots use actual values (i.e., historical data). The CM value for (1) is −0.060 indicating small instability with respect to MTTF and 0.087 for (2) indicating small stability with respect to normalized Total Failures. The corresponding standard deviations are 0.541 and 0.442. Large variability in CM is the case in this application due to the large variability in functionality across releases. Furthermore, it is not our objective to judge the process that is used in this example. Rather, our purpose in showing these and subsequent values of CM is to *illustrate* our model. We use these plots and the CM to assess the long-term stability

Fig. 2. Total failures per KLOC across releases.

of the maintenance process. We show example computations of CM for (1) and (2) in Table 3.

5.1.3 Total Test Time

We want Total Test Time, normalized by KLOC Change in Code, as computed by (3), to show a **decreasing** trend across releases, indicating that test effort is **decreasing** with respect to code changes.

$$\text{Total Test Time/KLOC} = \text{Total Test Time/KLOC Change in Code on the OI.} \qquad (3)$$

We plot (3) in Fig. 3 against Release Time of OI, using actual values. The CM value for this plot is 0.116, with a standard deviation of 0.626, indicating stability with respect to efficiency of test effort. We use this plot and the CM to assess whether testing is efficient with respect to the amount of code that has been changed.

5.2 Reliability Predictions

5.2.1 Total Failures

Up to this point, we have used only actual data in the analysis. Now we expand the analysis to use both predictions and actual data but only for the seven OIs where we could make predictions. Using the Schneidewind Model [1], [9], [16], [17], [18] and the SMERFS software reliability tool [4], we show prediction equations, using

30 day time intervals, and make predictions for OIA, OIB, OIC, OID, OIE, OIJ, and OIO. This model or any other applicable model may be used [1], [4].

To predict Total Failures in the range $[1, \infty]$ (i.e., failures over the life of the software), we use (4):

$$F(\infty) = \alpha/\beta + X_{s-1} \qquad (4)$$

where the terms are defined as follows:

s: starting time interval for using failures counts for computing parameters α and β,
α: initial failure rate,
β: rate of change of failure rate, and
X_{s-1}: observed failure count in the range $[1, s-1]$

Now, we predict Total Failures normalized by KLOC Change in Code. We want predicted normalized Total Failures to show a decreasing trend across releases. We computed a CM value for this data of 0.115, with a standard deviation of 0.271, indicating stability with respect to predicted normalized Total Failures.

5.2.2 Remaining Failures

To predict Remaining Failures $r(t)$ at time t, we use (5) [1], [9], [17]:

$$r(t) = F(\infty) - X_t \qquad (5)$$

TABLE 3
Example Computations of Change Metric (CM)

Operational Increment	MTTF (Days)	Relative Change	Total Failures/KLOC	Relative Change
A	179.7		0.750	
B	409.6	0.562	0.877	-0.145
C	406.0	-0.007	1.695	-0.483
D	192.3	-0.527	0.984	0.419
E	374.6	0.487	0.568	0.423
J	73.6	-0.805	0.238	0.581
O	68.8	-0.068	0.330	-0.272
	CM	-0.060	CM	0.087

Fig. 3. Total test time per KLOC across releases.
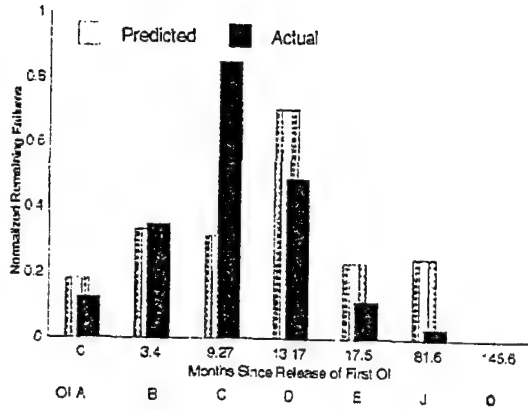
Fig. 4. Reliability of maintained software—remaining failures normalized by change to code.
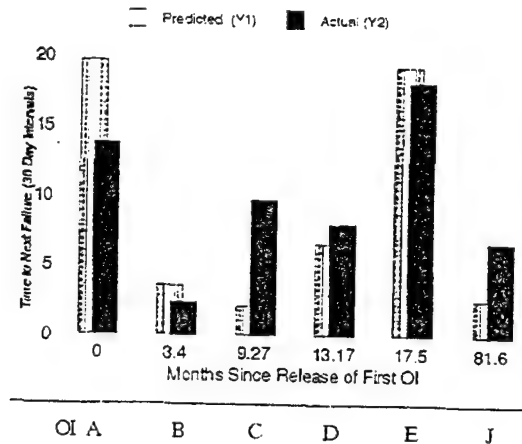
Fig. 5. Reliability of maintained software—time to next failure.

This is the predicted Total Failures over the life of the software minus the observed failure count at time t.

We predict Remaining Failures, normalize them by KLOC Change in Code, and compare them with normalized actual Remaining Failures for seven OIs in Fig. 4. We approximate Actual Remaining Failures at time t by subtracting the observed failure count at time t from the observed Total Failure count at time T, where $T \gg t$. The reason for this approach is that we are approximating the failure count over the life 'ty 12of the software by using the failure count at time T. We want (5) and actual Remaining Failures, normalized by KLOC Change in Code, to show a decreasing trend over a sequence of releases. The CM values for these plots are 0.107 and 0.277, respectively, indicating stability with respect to Remaining Failures. The corresponding standard deviations are 0.617 and 715.

5.2.3 Time to Next Failure

To predict the Time for the Next F_t Failures to occur, when the current time is t, we use (6) [1], [16], [17].

$$T_F(t) = [(\log \alpha/(\alpha - \beta(X_{s,t} + F_t))])/\beta] - (t - s - 1) \quad (6)$$

The terms in $T_F(t)$ have the following definitions:

t : Current time interval;

$X_{s,t}$: Observed failure count in the range $[s, t]$; and

F_t: Given number of failures to occur after interval t (e.g., one failure).

We want (6) to show an increasing trend over a sequence of releases. Predicted and actual values are plotted for six OIs (OIO has no failures) in Fig. 5. The CM values for these plots are -0.152 and -0.065, respectively, indicating slight instability with respect to time to next failure. The corresponding standard deviations are 0.693 and 0.630.

We predicted values of Total Failures, Remaining Failures, and Time to Next Failure as indicators of the risk of operating software in the future: Is the predicted future reliability of software an acceptable risk? The risk to the mission may or may be not be acceptable. If the latter, we take action to improve the maintained product or the maintenance process. We use actual values to measure the reliability of software and the risk of deploying it resulting from maintenance actions.

5.3 Summary

We summarize change metric values in Table 4. Overall (i.e., average CM), the values indicate marginal stability. If the majority of the results and the average CM were negative, this would be an alert to investigate the cause. The results could be caused by: 1) greater functionality and complexity in the software over a sequence of releases, 2) a maintenance process that needs to be improved, or 3) a combination of these causes.

TABLE 4
Change Metric Summary

Metric	Actual	Predicted
Mean Time To Failure	-0.060	
Total Test Time per KLOC	0.116	
Total Failures per KLOC	0.087	0.115
Remaining Failures per KLOC	0.277	0.107
Time to Next Failure	-0.065	-0.152
Average	**0.071**	
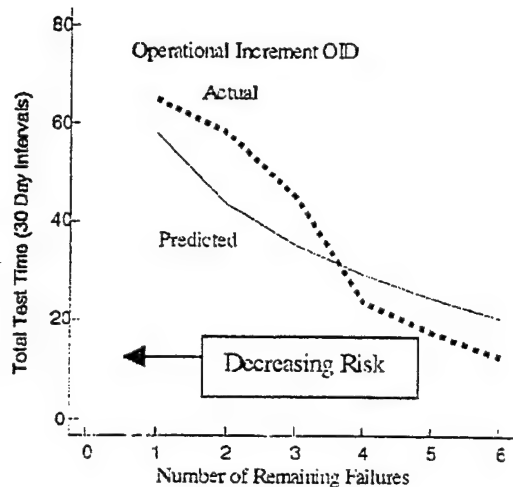
Fig. 6. Total test time to achieve remaining failures.

6 METRICS FOR LONG-TERM AND SHORT-TERM ANALYSIS

In addition to the long-term maintenance criteria, it is desirable that the maintenance effort results in increasing reliability within each release or OI. One way to evaluate how well we achieve this goal is to predict and observe the amount of test time that is required to reach a specified number of Remaining Failures. In addition, we want the test effort to be efficient in finding residual faults for a given OI. Furthermore, number of Remaining Failures serves as an indicator of the risk involved in using the maintained software (i.e., a high value of Remaining Failures portends a significant number of residual faults in the code). In the analysis that follows we use predictions and actual data for a selected OI to *illustrate* the process: OID.

6.1 Total Test Time Required for Specified Remaining Failures

We predict the Total Test Time that is required to achieve a specified number of Remaining Failures, $r(t_t)$, at time t_t, by (7) [1], [17]:

$$t_t = [\log [\alpha/(\beta [r(t_t)])]] /\beta + (s - 1) \quad (7)$$

We plot predicted and actual Total Test Time for OID in Fig. 6 against given number of Remaining Failures. The two plots have similar shapes and show the typical asymptotic characteristic of reliability (e.g., Remaining Failures) vs. Total Test Time. These plots indicate the possibility of big gains in reliability in the early part of testing; eventually the gains become marginal as testing continues. The figure also shows how risk is reduced with a decrease in Remaining Failures that is accomplished with increased testing. Predicted values are used to gauge how much maintenance test effort would be required to achieve desired reliability goals and whether the predicted amount of Total Test Time is technically and economically feasible. We use actual values to judge whether the maintenance test effort has been efficient in relation to the achieved reliability.
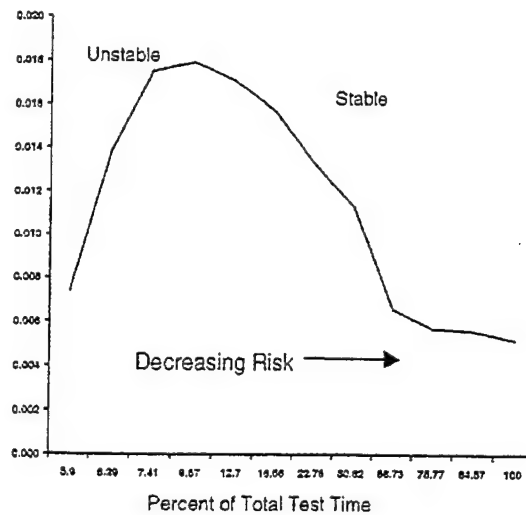
Fig. 7. OID failure rate.
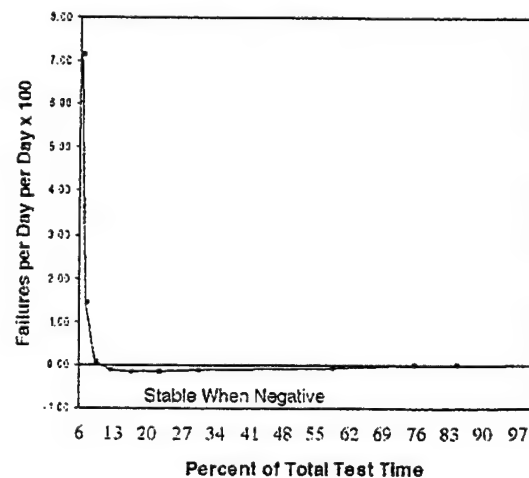
Fig. 8. OID rate of change of failure rate.
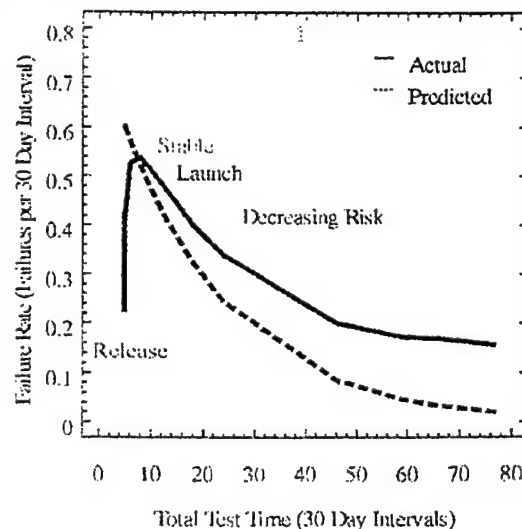
Fig. 9. OID failure rate predicted vs. actual.

183

6.2 Failure Rate

In the short term (i.e., within a release), we want the Failure Rate (1/MTTF) of an OI to **decrease** over an OI's Total Test Time, indicating **increasing** reliability. Practically, we would look for a decreasing trend, after an initial period of instability (i.e., increasing rate as personnel learn how to maintain new software). In addition, we use various shape metrics, as defined previously, to see how quickly we can achieve reliability growth with respect to test time expended. Furthermore, Failure Rate is an indicator of the risk involved in using the maintained software (i.e., an increasing failure rate indicates an increasing probability of failure with increasing use of the software).

$$\text{Failure Rate} = \text{Total Number of Failures During Test/Total Test Time} \quad (8)$$

We plot (8) for OID in Fig. 7 against Total Test Time since the release of OID. Fig. 7 *does* show that short-term stability is achieved (i.e., failure rate asymptotically approaches zero with increasing Total Test Time). In addition, this curve shows when the failure rate transitions from unstable (positive Failure Rate) to stable (negative Failure Rate). The figure also shows how risk is reduced with decreasing Failure Rate as the maintenance process stabilizes. Furthermore, in Fig. 8 we plot the rate of change (i.e., slope) of the Failure Rate of Fig. 7. This curve shows the percent of Total Test Time when the rate of change of Failure Rate reaches its maximum negative value. We use these plots to assess whether we have achieved short-term

TABLE 5
Percent of Total Test Time Required to Achieve Reliability Goals and Change Metrics (CM)

Operational Increment	One Remaining Failure (% Test Time)	Relative Change	Stable Failure Rate (% Test Time)	Relative Change	Maximum Failure Rate Change (% Test Time)	Relative Change
A	77.01		76.99		76.99	
B	64.11	0.168	64.11	0.167	64.11	0.167
C	32.36	0.495	10.07	0.843	10.07	0.843
D	84.56	-0.617	12.70	-0.207	22.76	-0.558
E	83.29	0.015	61.45	-0.793	61.45	-0.630
J	76.88	0.077	76.89	-0.201	76.89	-0.201
O	46.49	0.395	100.00	-0.231	100.00	-0.231
CM		0.089	CM	-0.070	CM	-0.101
STD DEV		0.392	STD DEV	0.543	STD DEV	0.544

TABLE 6
Shuttle Operational Increment Functonality

Operational Increment	Release Date	KLOC Change	Operational Increment Function
A	9/1/83	8.0	Redesign of Main Engine Controller.
B	12/12/83	11.4	Payload Re-manifest Capabilities.
C	6/8/84	5.9	Crew Enhancements.
D	10/5/84	12.2	Experimental Orbit Autopilot. Enhanced Ground Checkout.
E	2/15/85	8.8	Western Test Range. Enhance Propellant Dumps.
F	12/17/85	6.6	Centaur.
G	6/5/87	6.3	Post 51-L (Challenger) Safety Changes.
H	10/13/88	7.0	System Improvements.
I	6/29/89	12.1	Abort Enhancements.
J	6/18/90	29.4	Extended Landing Sites. Trans-Atlantic Abort Code Co-Residency.
K	5/21/91	21.3	Redesigned Abort Sequencer. One Engine Auto Contingency Aborts. Hardware Changes for New Orbiter.
L	6/15/92	34.4	Abort Enhancements.
M	7/15/93	24.0	On-Orbit Changes.
N	7/13/94	10.4	MIR Docking. On-Orbit Digital Autopilot Changes.
O	10/18/95	15.3	Three Engine Out Auto Contingency.
P	7/16/96	7.3	Performance Enhancements.
Q	3/5/97	11.0	Single Global Positioning System.

TABLE 7
Chronology of Process Improvements

Year in which Process Improvement Introduced	Process Improvement
1976	Structured Flows
1977	Formal Software Inspections
1978	Formal Inspection Moderators
1980	Formalized Configuration Control
1981	Inspection Improvements
1982	Configuration Management Database
1983	Oversight Analyses
1984	Build Automation Formalized Requirements Analysis
1985	Quarterly Quality Reviews Prototyping
1986	Inspection Improvements Formal Requirements Inspections
1987	Process Applied to Support Software
1988	Reconfiguration Certification Reliability Modeling and Prediction
1989	Process Maturity Measurements
1990	Formalized Training
1992	Software Metrics

stability in the maintenance process (i.e., whether Failure Rate decreases asymptotically with increasing Total Test Time). If we obtain contrary results, this would be an alert to investigate whether this is caused by: 1) greater functionality and complexity of the OI as it is being maintained, 2) a maintenance process that needs to be improved, or 3) a combination of these causes.

Another way of looking at failure rate with respect to stability and risk is the annotated Failure Rate of OID shown in Fig. 9, where we show both the actual and predicted Failure Rates. We use (8) and (9) [1] to compute the actual and predicted Failure Rates, respectively, where i is a vector of time intervals for $i \geq s$ in (9).

$$f(i) = \alpha(\text{EXP}(-\beta(i - s + 1))) \qquad (9)$$

A 30-day interval has been found to be convenient as a unit of NASA Space Shuttle test time because testing can last for many months or even years. Thus, this is the unit used in Fig. 9, where we show the following events in

intervals, where the predictions were made at 12.73 intervals:

Release time: 0 interval,

Launch time: 13.90 intervals,

Predicted time of maximum Failure Rate: 6.0 intervals,

Actual time of maximum Failure Rate: 7.43 intervals,

Predicted maximum Failure Rate: 0.5735 failures
   per interval, and

Actual maximum Failure Rate: 0.5381 failures per interval.

In Fig. 9, stability is achieved after the maximum failure rate occurs. This is at $i = s$ (i.e. $i = 6$ intervals) for predictions because (9) assumes a monotonically decreasing failure rate, whereas the actual failure rate increases, reaches a maximum at 7.43 intervals, and then decreases. Once stability is achieved, risk decreases.

185

6.3 Summary

In addition to analyzing short-term stability with these metrics, we use them to analyze long-term stability across releases. We show the results in Table 5 where the percent of Total Test Time to achieve reliability growth goals is tabulated for a set of OIs, using actual failure data, and the Change Metrics are computed. Overall, the values of CM indicate marginal instability. Interestingly, except for OID, the maximum negative rate of change of failure rate occurs when Failure Rate becomes stable, suggesting that maximum reliability growth occurs when the maintenance process stabilizes.

7 SPACE SHUTTLE OPERATIONAL INCREMENT FUNCTIONALITY AND PROCESS IMPROVEMENT

Table 6 shows the major functions of each OI [12] along with the Release Date and KLOC Change repeated from Table 1 and Table 2. There is a not a one-for-one relationship between KLOC Change and the functionality of the change because, as stated earlier, KLOC is an indicator of maintenance actions, not functionality. However, the software developer states that there has been increasing software functionality and complexity with each OI, in some cases with *less* rather than more KLOC [8]. The focus of the early OIs was on launch, orbit, and landing. Later OIs, as indicated in Table 6, built upon this baseline functionality to add greater functionality in the form of MIR docking and the Global Positioning System (GPS), for example. Table 7 shows the process improvements that have been made over time on this project, indicating continuous process improvement across releases.

The stability analysis that was performed yielded mixed results: About half are favorable and half are unfavorable. Some variability in the results may be due to gaps in the data caused by OIs that have experienced insufficient failures to permit statistical analysis. Also, we note that the values of CM are marginal for both the favorable and unfavorable cases. Although there is not pronounced stability neither is there pronounced instability. If there were consistent and large negative values of CM, it would be cause for alarm and would suggest the need to perform a thorough review of the process. This is not the case for the NASA Space Shuttle. We suspect, but cannot prove, that in the absence of the process improvements of Table 7 the CM values would look much worse. It is very difficult to associate a specific product improvement with a specific process improvement. A controlled experiment would be necessary to hold all process factors constant and observe the one factor of interest and its influence on product quality. This is infeasible to do in industrial organizations. However, we suggest that in the *aggregate* a series of process improvements is beneficial for product quality and that a *set* of CM values can serve to highlight possible process problems.

8 Conclusions

As stated in the Introduction, the authors' emphasis in this paper was to propose a unified product and process measurement model for both product evaluation and process stability analysis. We were less interested in the results of the NASA Space Shuttle stability analysis, which was used to illustrate the model concepts. The authors concluded, based on both predictive and retrospective use of reliability, risk, and test metrics, that it is feasible to measure and assess both product quality and the stability of a maintenance process. The model is not domain specific. Different organizations may obtain different numerical results and trends than the ones we obtained for the NASA Space Shuttle.

REFERENCES

[1] "Recommended Practice for Software Reliability, R-013-1992," Am. Nat'l Standards Inst./Am. Inst. of Aeronautics and Astronautics, 1993.

[2] C. Billings et al., "Journey to a Mature Software Process," IBM Systems J., vol. 33, no. 1. pp. 46-61, 1994.

[3] L.C. Briand, V.R. Basili, and Y.-M. Kim, "Change Analysis Process to Characterize Software Maintenance Projects," Proc. Int'l Conf. Software Maintenance, pp. 38-49, Victoria, B.C., Canada, Sept. 1994.

[4] W.H. Farr and O.D. Smith, "Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide," NAVSWC TR-84-373, rev. 3, Naval Surface Weapons Center, Sept. 1993.

[5] D. Gefen and S.L. Schneberger, "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications," Proc. Int'l Conf. Software Maintenance. pp. 134-141, Monterey, Calif., Nov. 1996.

[6] J. Henry, S. Henry, D. Kafura, and L. Matheson, "Improving Software Maintenance at Martin Marietta," IEEE Software, vol. 11, no. 4, pp. 67-75, July 1994.

[7] C. Hollenbach et al., "Combining Quality and Software Improvement," Comm. ACM, vol. 40. no. 6, pp. 41-45, June 1997.

[8] Private Communication with Ted Keller, IBM, Apr. 1998.

[9] T. Keller, N.F. Schneidewind, and P.A. Thornton, "Predictions for Increasing Confidence in the Reliability of the Space Shuttle Flight Software," Proc. AIAA Computing in Aerospace 10, pp. 1-8, San Antonio, Tex., Mar. 1995.

[10] T.M. Khoshgoftaar, E.B. Allen, R. Halstead, and G.P. Trio, "Detection of Fault-Prone Software Modules during a Spiral Life Cycle," Proc. Int'l Conf. Software Maintenance, pp. 69-76, Monterey, Calif., Nov. 1996.

[11] M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," Proc. IEEE, vol. 68, no. 9, Sept. 1980.

[12] "Software Release Schedules," Lockheed Martin, Jan. 1998.

[13] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities," Proc. Int'l Conf. Software Maintenance, pp. 295-303, Opio (Nice), France, Oct. 1995.

[14] T.M. Pigoski and L.E. Nelson, "Software Maintenance Metrics: A Case Study," Proc. Int'l Conf. Software Maintenance, pp. 392-401, Victoria, BC Canada, Sept. 1994.

[15] N.F. Schneidewind, "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics," Proc. Int'l Conf. Software Maintenance, pp. 232-239, Bari, Italy, Oct. 1997.

[16] N.F. Schneidewind, "Reliability Modeling for Safety Critical Software," IEEE Trans. Reliability, vol. 46, no. 1, pp. 88-98, Mar. 1997.

[17] N.F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data," IEEE Trans. Software Eng., vol. 19, no. 11, pp. 1,095-1,104, Nov. 1993.

[18] N.F. Schneidewind and T.W. Keller, "Application of Reliability Models to the Space Shuttle," *IEEE Software*, vol. 9, no. 4, pp. 28-33, July 1992.

[19] H. Sneed, "Modelling the Maintenance Process at Zurich Life Insurance," *Proc. Int'l Conf. Software Maintenance*, pp. 217-226, Monterey, Calif., Nov. 1996.

[20] G.E. Stark, "Measurements for Managing Software Maintenance," *Proc. Int'l Conf. Software Maintenance*, pp. 152-161, Monterey, Calif., Nov. 1996.

Norman F. Schneidewind is a professor of information sciences and director of the Software Metrics Research Center at the Naval Postgraduate School. He is the developer of the Schneidewind software reliability model that has been used by NASA to assist in the prediction of software reliability of the Space Shuttle—one of the models recommended by the American National Standards Institute (ANSI) and the American Institute of Aeronautics and Astronautics Recommended Practice for Software Reliability. He received an award for outstanding research achievements by the Naval Postgraduate School in 1992 and 1998. He received the IEEE Computer Society's Outstanding Contribution Award for work leading to the establishment of IEEE Standard 1061-1992 in 1993. In addition, he received the IEEE Computer Society Meritorious Service Award for his long-term committed work in advancing the cause of software engineering standards. He was recognized for his contributions to the IEEE Computer Society by being named to the Golden Core of volunteers. Dr. Schneidewind is a fellow of the IEEE, elected for contributions to software measurement models in reliability and metrics and for leadership in advancing the field of software maintenance.

Cost Framework for COTS Evaluation

Norman F. Schneidewind
Naval Postgraduate School
Email: nschneid@nps.navy.mil

Cost as the Universal COTS Metric

We focus on factors that the user should consider when deciding whether to use COTS software. We take the approach of using the common denominator *cost*. This is done for two reasons: First, cost is obviously of interest in making such decisions and second a single metric – cost in dollars – can be used for evaluating the pros and cons of using COTS. The reason is that various software system attributes, like acquisition cost and availability (i.e., the percentage of scheduled operating time that the system is available for use), are non-commensurate quantities. That is, we cannot relate quantitatively "a low acquisition cost" with "high availability". These units are neither additive nor multiplicative. However, if it were possible to translate availability into either a cost gain or loss for COTS software, we could operate on these metrics mathematically. Naturally, in addition to cost, the user application is key in making the decision. Thus one could develop a matrix where one dimension is *application* and the other dimension is the various *cost elements*. We show how cost elements can be identified and how cost comparisons can be made over the *life* of the software. Obviously, identifying the costs would not be easy. The user would have to do a lot of work to set up the decision matrix but once it was constructed, it would be a significant tool in the evaluation of COTS. Furthermore, even if all the required data cannot be collected, having a framework that defines software system attributes would serve as a user guide for factors to consider when making the decision about whether to use COTS software or in-house developed software.

Certainly, different applications would have varying degrees of relationships with the cost elements. For example, flight control software would have a stronger relationship with the cost of unavailability than a spreadsheet application. Conversely, the latter would have a stronger relationship with the cost of inadequacy of tool features than the former. Due to the difficulty of identifying specific COTS-related costs, our initial approach is to identify cost elements on the ordinal scale. Thus, the first version of the decision matrix would involve ordinal scale metrics (i.e., the cost of unreliability is more important for flight control software than for spreadsheet applications). As the field of COTS analysis matures and as additional data is collected about the cost of using COTS, we will be able to refine our metrics to the ratio scale (e.g., the cost of unreliability in COTS systems is two times that in custom systems).

The cost elements for comparing COTS software with in-house software are identified below. This list is not exhaustive; its purpose is to illustrate the approach. These elements apply whether we are comparing a system comprised of all COTS components with all in-house components or comparing only a subset of COTS components with corresponding in-house components. Explanatory comments are made where necessary. Mean values are used for some quantities in the initial framework. This is the case because it will be a challenge to collect *any* data for some applications. Therefore, the initial framework should not be overly complex. Variance and statistical distribution information could be included as enhancements if the initial framework proves successful.

Cost Elements

$C_c(j)$ = Cost of acquiring COTS software in year j.

$C_i(j)$ = Cost of developing in-house software in year j.

$U_c(j)$ = Cost of upgrading COTS software in year j.

$U_i(j)$ = Cost of upgrading in-house software in year j.

$P(j)$ = Cost of personnel who use the software system in year j. This quantity represents the value to the customer of using the software system.

$M_c(j)$ = Cost per unit time of repairing a fault in COTS software in year j. This is the cost of customer time involved in resolving a problem with the vendor.

$M_i(j)$ = Cost per unit time of repairing a fault in in-house software in year j.

$R_c(j)$ = Mean time of repairing a fault that causes a failure in COTS software in year j. This is the average time that the user spends in resolving a problem with the vendor.

$R_i(j)$ = Mean time of repairing a fault that causes a failure in in-house software in year j.

$T(j)$ = Scheduled operating time for the software system in year j.

$A_c(j)$ = Availability of software system that uses COTS software in year j.

$A_i(j)$ = Availability of software system that uses software developed in-house in year j.

These quantities are the fractions of $T(j)$ that the software system is available for use.

$F_c(j)$ = Failure rate of COTS software in year j.

$F_i(j)$ = Failure rate of COTS software in year j.

These quantities are the number of failures per year that cause loss of productivity and availability of the software system.

In some applications, some or all of the above quantities may be known or assumed to be constant over the life of the software system. Using the above cost elements, we derive the equations for the annual costs of the two systems and the difference in these costs. In the cost difference calculations that follow, a positive quantity is favorable to in-house development and a negative quantity is favorable to COTS.

Cost of Acquiring Software

Difference in annual cost = $C_c(j) - C_i(j)$ (1)

Cost of Upgrading Software

Difference in annual cost = $U_c(j) - U_i(j)$ (2)

Cost of Software being Unavailable for Use

Annual cost of COTS software being unavailable for use
= $(1-A_c(j)) * P(j)$.

Annual cost of the in-house software being unavailable for use
= $(1-A_i(j)) * P(j)$.

Difference in annual cost = $P(j) * (A_i(j) - A_c(j))$ (3)

Cost of Repairing Software

Average annual cost of repairing failed COTS software = $F_c(j) * T(j) * R_c(j) * M_c(j)$.

Average annual cost of repairing failed in-house software
= $F_i(j) * T(j) * R_i(j) * M_i(j)$.

Difference in annual cost =
$T(j) * ((F_c(j) * R_c(j) * M_c(j)) - ((F_i(j) * R_i(j) * M_i(j)))$ (4)

Then, TC_j, total difference in cost in year j, is the sum of (1), (2), (3), and (4). Because there is the opportunity to invest funds in alternate projects, costs in different years are not equivalent (i.e., funds available today have more value than an equal amount in the future because they could be invested today and earn a future return). Therefore, a stream of costs over the life of the software for n years must be discounted by k, the rate of return on alternate use of funds. Thus the total discounted cost differential between COTS software and in-house software is:

$\sum_i^n TC_i / (1+k)^i$

In this initial formulation, we have not included possible differences in functionality between the two approaches. However, a reasonable assumption is that COTS software would not be considered unless it could provide minimum functionality to satisfy user requirements. Thus, a typical decision for the user is whether it is worth the additional life cycle costs to develop an in-house software system with all the desirable attributes.

Predicting Deviations in Software Quality by Using Relative Critical Value Deviation Metrics

Norman F. Schneidewind, Ph.D.
Division of Computer and Information
Sciences and Operations
Naval Postgraduate School
Monterey, CA 93943
Email: nschneid@nps.navy.mil

Allen P. Nikora, Ph.D.
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Email: Allen.P.Nikora@jpl.nasa.gov

Abstract

We develop a new metric, Relative Critical Value Deviation (RCVD), for classifying and predicting software quality. The RCVD is based on the concept that the extent to which a metric's value deviates from its critical value, normalized by the scale of the metric, indicates the degree to which the item being measured does not conform to a specified norm. For example, the deviation in body temperature above 98.6 Fahrenheit degrees is a surrogate for fever. Similarly, the RCVD is a surrogate for the extent to which the quality of software deviates from acceptable norms (e.g., zero discrepancy reports). Early in development, surrogate metrics are needed to make predictions of quality before quality data are available. The RCVD can be computed for a single metric or multiple metrics. Its application is in assessing newly developed modules by their quality in the absence of quality data. The RCVD is a part of the larger framework of our measurement models that include the use of Boolean Discriminant Functions for classifying software quality. We demonstrate our concepts using Space Shuttle flight software data.

Keywords: Quality classification and prediction, relative critical value deviation metrics.

1. Introduction

Our goal is to provide models and processes to assist software managers in answering the following questions:

- How can I control the quality of my software?
- How can I predict the quality of my software?
- How shall I prioritize my effort to achieve my quality goals?
- How can I determine whether my quality goals are being met?
- How much will it cost to achieve my quality goals?

We develop quality control and prediction models that are used to identify modules requiring priority attention during development and maintenance. This is accomplished in two activities: *validation* and *application*. During *validation*, we use a build of the software that has been developed as the source of data to compute Boolean Discriminant Functions (BDFs), Relative Critical Value Deviation (RCVD) metrics, and regression equations that we use to retrospectively classify and predict quality with specified accuracy, by build and module. Using these functions and equations during *application*, we classify and predict the quality of new software that is being developed. This is the quality we expect to achieve during maintenance. During *validation*, both quality factor (e.g., discrepancy reports of deviations between requirements and implementation) and software metrics (e.g., size, structural) data are available; during *application*, only the latter are available. During *validation*, we construct Boolean discriminant functions (BDFs) comprised of a set of metrics and their critical values (i.e., thresholds) [1, 2]. We select the best BDF based on its ability to achieve the maximum relative incremental quality/cost ratio. During *application*, if at least one of the module's metrics has a value that exceeds its critical value, the module is identified as "high priority" (i.e., low quality); otherwise, it is identified as "low priority" (i.e., high quality). Our objective is to identify and correct quality problems during development, as opposed to waiting until maintenance when the cost of correction would be high. This process addresses the question: "How can I control the quality of my software?" Because BDFs only provide an *accept/reject* decision on module quality, during validation, we also construct RCVDs that are used to prioritize the effort applied to rejected modules. In other words, an RCVD measures the *degree* to which quality is low. This process addresses the question: "How shall I prioritize my effort to achieve my quality goals?

A RCVD is a derived metric, based on the normalized deviation between a metric's value and its critical value. It may be based on a single or multiple metrics. In our process, we: 1) identify the critical values of the metrics and 2) find the optimal BDF and RCVD based on their ability to

190

satisfy both *statistical* and *application* criteria. Statistical criteria refer to the ability to correctly classify the software (i.e., classify high quality software as high quality and low quality software as low quality). Application criteria refer to the ability to achieve a high quality/cost ratio. This process addresses the questions: "How can I determine whether my quality goals are being met?" and "How much will it cost to achieve my quality goals?"

RCVD values that exceeded the .80 percentile value were able to account for two-thirds of the discrepancy reports. To round out our approach, we use regression equations to predict quality limits. This is desirable because, although BDFs and RCVDs control and predict quality based on expected values, they are not capable of predicting the range of quality values.

We show that it is important to perform a marginal analysis (i.e., identification of the incremental contribution of each metric to improving quality) when making a decision about how many metrics to include in the BDFs and RCVDs. If many metrics are added to the set at once, the contribution of individual metrics is obscured. Also, the marginal analysis provides an effective rule for deciding when to stop adding metrics.

The contributions of this research are the following: 1) the Relative Critical Value Deviation (RCVD) is a new metric for classifying and predicting software quality; 2) the RCVDs in combination with the BDFs we previously developed, allow the software manager to both control quality and prioritize the effort required to achieve quality goals; 3) BDFs, RCVDs, and regression equations are *integrated* into a process to assist the software manager in answering the questions posed in the introduction; and 4) the data and most of the calculations are implemented in a spreadsheet for easy transfer to practitioners.

1.1 Related Research

Our models are in the class of models concerned with the classification, control, and prediction of quality. Other researchers have had similar objectives but different approaches. Porter and Selby used classification trees to partition multiple metric value space so that a sequence of metrics and their critical values could be identified that were associated with either high quality or low quality software [3]. This technique is closely related to our approach of identifying a set of metrics and their critical values that will satisfy quality and cost criteria. However, we use statistical analysis to make the identification.

Briand et al. used logistic regression to classify modules as fault-prone or not fault-prone as a function of various object oriented metrics [4]. In another example of logistic regression, Khoshgoftaar and Allen used it to classify modules as fault-prone or not fault-prone as a function of faults, requirements, performance, and documentation software trouble report metrics [5]. While one of our objectives is similar -- classify modules as either high quality or low quality -- we derive from this *binary* classification

several predictive *continuous* quality and cost metrics, including the RCVDs. These metrics are used to predict the quality of software that will be delivered by development to maintenance and the cost of achieving it.

Khoshgoftaar et al. used nonparametric discriminant analysis in each iteration of a military system project to predict fault-prone modules in the next iteration [6]. This approach provided early indication of reliability and the risk of implementing the next iteration. They conducted a similar study involving a telecommunications application, again using nonparametric discriminant analysis, to classify modules as either fault-prone or not fault-prone [7]. Our approach has the same objective but we produce BDFs and RCVDs in terms of the original metrics as opposed to using density functions as discriminators.

Khoshgoftaar and Allen have also developed models for ranking modules for reliability improvement according to their degree of fault-proneness as opposed to whether they are fault-prone or not [8]. They used Alberg Diagrams [9] that predict percentage of faults as a function of percentage of modules by ordering modules in decreasing order of faults and noting the cumulative number of faults corresponding to various percentages of modules. Our approach is similar but we accomplish the same objective by sorting the modules by RCVD and finding its percentile distribution and the corresponding *drcount* percentile distribution, as we explain later.

2. Discriminative Power Model

2.1. Discriminative Power Validation

Using our metrics validation methodology [10, 11], and the *Space Shuttle* flight software metrics and discrepancy reports (DRs), we validate metrics with respect to the quality factor *drcount*. This is the number of discrepancy reports written against a module. In brief, this involves conducting statistical tests to determine whether there is a high degree of association between *drcount* and candidate metrics. As shown in Figure 1, we validate metrics on Build 1 (1397 modules) and apply them to Build 2 (846 modules) of the *Space Shuttle* flight software. Nikora and Munson argue for the need of a measurement baseline against which evolving systems may be compared [12]. Our baseline is Build 1 in Figure 1. The measurement results from Build 1 provide the data source for controlling and predicting the quality delivered to maintenance and for comparing predicted with actual quality, once the latter is known. Next, we define *Discriminative Power*.

2.1.1. Discriminative Power

Given the elements M_{ij} of a matrix of **n** modules and **m** metrics (i.e., **nm** metric values), the elements MC_j of a vector of **m** metric critical values, the elements F_i of a vector of **n** quality factor values, and scalar FC of quality

factor critical value, M_{ij} must be able to discriminate with respect to F_i, for a specified FC, as shown below:

$$M_{ij} > M_j \leftrightarrow F_i > FC \text{ and } M_{ij} \leq M_j \leftrightarrow F_i \leq FC \quad (1)$$

for i=1,2,...,n, and j=1,2,...,m with specified α, where α is the significance level of various statistical tests that are used for estimating the degree to which a set of metrics can correctly classify software quality. In other words, do the indicated metric relations imply corresponding quality factor relations in (1)? This criterion assesses whether MC_j has sufficient *Discriminative Power* to be capable of distinguishing a set of high quality modules from a set of low quality modules. If so, we use the critical values in Quality Control and Prediction described below. The validation process is illustrated in Figure 1, where the critical values MC_j are produced during the Test phase of Build 1 by using the metrics M_{ij} from the Design phase and the quality factor F_i (e.g., *drcount*) available in the Test phase. (Discrepancy Reports are written against the software throughout development but they are not significantly complete until the end of the Test phase during which failures are observed). The desired quality level is set by the choice of FC. The lower its value, the higher the quality requirement; conversely, the higher its value, the lower the requirement. A value of zero is appropriate for safety-critical systems like the *Space Shuttle*.

2.2. Relative Critical Value Deviation (RCVD) Metric

The RCVD is based on the concept that the extent to which a metric's value deviates from its critical value, normalized by the scale of the metric, is an indicator of the degree to which the entity being measured does not conform to a specified norm. For example, the extent to which body temperature exceeds 98.6 degrees Fahrenheit is an indicator of the deviation from an established norm of human health. Measurement involves using surrogates: the deviation in temperature above 98.6 degrees is a surrogate for fever. Similarly, the RCVD is a surrogate for the extent that software quality deviates from acceptable norms (e.g., zero discrepancy reports). The concept of the RCVD is shown in Figure 2, where the metric and quality scales are shown, defined by the maximum (MX_{ij} and minimum (MN_j) metric boundaries and the maximum (FX) and minimum (FN) quality boundaries, respectively. The theory of the RCVD is given by the following relation:

$$RCVD_{ij} = (M_{ij} - MC_j)/(MX_j - MN_j) \leftrightarrow (F_i - FC)/(FX - FN) \quad (2)$$

This means that the deviation of a metric from its critical value, normalized by metric length, is related to the degree of quality, as represented by the normalized deviation of a quality factor (e.g., *drcount*) from its critical values: increasing positive deviations are related to decreasing quality and increasing negative deviations are related to increasing quality. It should not be inferred that

the relationship is linear or proportional; in fact, it is non-linear. In the idealized diagram in Figure 2, the worst quality corresponds to MX_j and FX, the best quality to MN_j and FN, and acceptable quality to MC_j and FC. Also, Figure 2 does not indicate the mathematical form of F_i. If FN is equal to zero and F_c is set equal to zero, which is frequently the case, F_i and FX can be replaced by the sum of the quality factor across a set of modules and the total quality factor, respectively. This quantity is the proportion of *drcount* computed across a set of modules. An RCVD can also be comprised of multiple metrics by computing their mean. Note that although it would not be valid to compute the mean of metrics, the mean of RCVDs is another story since these are normalized dimensionless quantities. We experimented with both single and multiple metric RCVDs, as we explain later.

2.3. Quality Control and Prediction

Quality control is the evaluation of modules with respect to predetermined critical values of metrics. The purpose of quality control is identify software that does not meet quality requirements early in the development process so corrective action can be taken when the cost is low. Quality control is applied during the Design phase of Build 2 in Figure 1 to flag software for detailed inspection that is below quality limits. The validated BDFs, comprised of the metrics M_{ij} and their critical values MC_j that are obtained from Build 1, are used to either accept or reject the modules of Build 2 [1, 2]. At this point during the development of Build 2, only the metric data M_{ij} and MC_j are available. The validated RCVDs are used to prioritize the attention and effort devoted to modules that are rejected by the BDFs. Details are given later.

Quality predictions are used by the developer to anticipate rather than react to quality problems. Figure 1 shows the metrics controlling and predicting the quality of software that will be delivered to maintenance *early* in the development of Build 2. Accompanied by rigorous inspection and test, this process will result in improved quality of Build 2 and the software that is released to maintenance. Once all of the quality factor data F_i (e.g., *drcount*) have been collected for Build 2, at the end of the Test phase as shown in Figure 1, the quality of Build 2 would be known. This, then, becomes the actual quality of Build 2 in the maintained software. Regression equations $F_i=f(M_{ij})$ are developed during the Test phase of Build 1 and applied to predicting quality limits during the Design Phase of Build 2, as shown in Figure 1. This process addresses the question: "How can I predict the quality of my software?"

3. Validation Methodology

We use a five stage process to select metrics and metric functions for quality control and prediction: 1) com-

pute critical values of the candidate metrics; 2) for the set of candidate metrics and critical values, find the optimal BDF based on statistical and application criteria; 3) apply a stopping rule for adding metrics; 4) identify the best RCVD for prioritizing quality assurance effort; and 5) develop a regression equation that will accurately predict quality limits (e.g., limits of *drcount*). Table 1 provides a functional description of each stage. The five stages take place during the Test Phase of Build 1 of Figure 1, once all the quality factor data F_i (e.g., *drcount*) are available. The next sections describe the analysis for each stage.

3.1. Stage 1: Compute Critical Values

Critical values MC_j are computed based on the Kolmogorov-Smirnov (K-S) test [1, 2]. Table 1 shows the metric definitions, critical values MC_j, and K-S distances for six metrics of Build 1. These metrics were selected based on their relatively high K-S distance compared to other metrics that had been collected on the *Space Shuttle*. The test statistic is the maximum vertical difference between the CDFs of two complementary sets of data (e.g., the CDFs of M_{ij} for $drcount \leq FC$ and $drcount > FC$). If the difference is significant (i.e., $\alpha \leq .005$), the value of M_{ij} corresponding to maximum CDF difference is used for MC_j. This relationship is expressed in equation (3). Metrics are added to the BDF in order of their K-S Distance.

$$K - S\ (MC_i) = \max \{[CDF\ (M_{ij}/(F_i \leq FC))] - [CDF\ (M_{ij}/(F_i > FC))]\} \quad (3)$$

3.2. Stage 2: Form a Set of Boolean Discriminate Functions (BDFs)

For each BDF identified in Stage 1 we use Table 2 to further evaluate the ability of the functions to discriminate high quality from low quality, from both statistical (e.g., misclassification rates) and application (e.g., ability of the metric set to correctly classify low quality modules) standpoints. In Table 2, MC_j and FC classify modules into one of four categories. The left column contains modules where none of the metrics exceeds its critical value; this condition is expressed with a Boolean AND function of the metrics. This is the *ACCEPT* column, meaning that according to the classification decision made by the metrics, these modules have acceptable quality. The right column contains modules where at least one metric exceeds its critical value; this condition is expressed by a Boolean OR function of the metrics. This is the *REJECT* column, meaning that according to the classification decision made by the metrics, these modules have unacceptable quality. The top row contains modules that are high quality; these modules have a quality factor that does not exceed its critical value (e.g., *drcount*=0). The bottom row contains modules that are low quality; these modules have a quality factor that exceeds its critical value (e.g., *drcount*>0).

Equation (4) gives the algorithms for making the cell counts, using the BDFs of F_i and M_{ij} that are calculated over the n modules for m metrics. This equation is an implementation of the relation given in (1).

$$C_{11} = \text{COUNT}_{i=1}^{n}\ \text{FOR}\ ((F_i \leq FC) \wedge (M_{i1} \leq MC_1)...\wedge (M_{ij} \leq MC_j)...\wedge (M_{im} \leq MC_m))$$

$$C_{12} = \text{COUNT}_{i=1}^{n}\ \text{FOR}\ ((F_i \leq FC) \wedge ((M_{i1} > MC_1)...\vee (M_{ij} > MC_j)...\vee (M_{im} > MC_m)))$$

$$(4)$$

$$C_{21} = \text{COUNT}_{i=1}^{n}\ \text{FOR}\ ((F_i > FC) \wedge (M_{i1} \leq MC_1)...\wedge (M_{ij} \leq MC_j)...\wedge (M_{im} \leq MC_m))$$

$$C_{22} = \text{COUNT}_{i=1}^{n}\ \text{FOR}\ ((F_i > FC) \wedge ((M_{i1} > MC_1)...\vee (M_{ij} > MC_j)...\vee (M_{im} > MC_m)))$$

for j=1,...,m, and where COUNT(i)=COUNT(i-1)+1 FOR Boolean expression *true* and COUNT(i)=COUNT(i-1), otherwise; COUNT(0)=0. The counts (C_{11}, C_{12}, C_{21}, and C_{22}) correspond to the cells of Table 2, where row and column totals are also shown: n, n_1, n_2, N_1, and N_2.

In addition to counting modules in Table 2, we must also count the quality factor (e.g., *drcount*) that is incorrectly classified. This is shown as Remaining Factor, RF, in the *ACCEPT* column. This is the quality factor count on modules that should have been rejected. Also shown is Total Factor, TF, the total quality factor count on all the modules in the build. Table 2 and subsequent equations show an example validation, where the combination of metrics from Table 1 and their critical values for Build 1 is *prologue size* (P) with a critical value of 63, *statements* (S) with a critical value of 27, and *eta2* (E2) with a critical value of 45. This is the optimal BDF. Later we will explain how we arrived at this particular combination of metrics as the optimal set. The results of the following calculations for the optimal BDF are shown in Table 3.

3.2.1. Statistical Criteria

We validate a BDF statistically by demonstrating that it partitions Table 2 so that C_{11} and C_{22} are large relative to C_{12} and C_{21}. If this is the case, a large number of high quality modules (e.g., modules with *drcount*=0) would have $M_{ij} \leq MC_j$ and would be correctly classified as high quality. Similarly, a large number of low quality modules (e.g., modules with *drcount*>0) would have $M_{ij} > MC_j$ and would be correctly classified as low quality. We evaluate partitioning ability using the misclassification rates.

3.2.2. Misclassification

We compute the degree of misclassification in Table 2 by noting that ideally $C_{11} = n_1 = N_1$, $C_{12} = 0$, $C_{21} = 0$, $C_{22} = n_2 = N_2$. The extent to which this is not the case is estimated by *Type 1* misclassifications (i.e., the module has *Low Quality* and the metrics "say" it has *High Quality*) and *Type 2* misclassifications (i.e., the module has *High Quality* and the metrics "say" it has *Low Quality*). Thus, we define the following measures of misclassification:

193

Proportion of Type 1 : $P_1 = C_{21}/n$
For the example, $P_1 = (35/1397)*100 = 2.51\%$ (5)

Proportion of Type 2 : $P_2 = C_{12}/n$
For the example, $P_1 = (344/1397)*100 = 24.62\%$ (6)

3.2.3. Application Criteria

Because it is the performance of the metrics in the application context that counts, we also validate metrics with respect to the application criteria *Quality and Inspection*, which are related to quality achieved and the cost to achieve it, respectively [1, 2]. During the Design phase of Build 2 in Figure 1, we predict that the quality computed by equations (7)--(9) will be delivered to maintenance, assuming that the modules rejected by the quality control process are inspected and tested and that the problems that are found are corrected. Furthermore, we predict that the degree of inspection computed by equation (10) will be required to achieve this quality. In addition to controlling and predicting quality, equations (7)--(9) can be used to address the question: "How can I determine whether my quality goals are being met?" For example, if a quality goal is ≤3% residual defects, the achievement of this goal can be measured by RFP -- equation (9). Also, the degree of rigorous inspection -- equation (10) can be used to address the question: "How much will it cost to achieve my quality goals?"

3.2.4. Quality

First, we estimate the metrics' ability to correctly classify quality, given that the quality is known to be low:

LQC : proportion of low quality (e.g., drcount > 0)
software correctly classified $= C_{22}/n_2$ (7)

For the example, LQC=(541/576)*100=93.92%.

Second, we estimate the metrics' ability to correctly classify quality, given that the BDF has classified modules as *ACCEPT*. This is done by summing quality factor in the *ACCEPT* column in Table 2 to produce Remaining Factor, RF (e.g., remaining *drcount*), given by equation (8).

$$RF = \sum_{i=1}^{n} F_i \text{ FOR } ((F_i > FC) \wedge (M_{i1} \le MC_1)...\wedge$$
$$(M_{ij} \le MC_j)...\wedge (M_{im} \le MC_m))$$ (8)

for j=1,...,m. This is the sum of F_i (e.g., *drcount*) on modules incorrectly classified as high quality because, for these modules, $(F_i > FC) \wedge (M_{ij} \le MC_j)$.

We estimate the proportion of RF by equation (9), where TF is the total F_i for the build.

$$RFP = RF/TF$$ (9)

For the example, from Table 2 there are 56 DRs on 35 modules that are incorrectly classified (i.e., RF=56). The total number of DRs for the 1397 modules is 2579. Therefore, RFP=(56/2579)*100=2.17%.

3.2.5. Inspection

Inspection is one of the costs of high quality. We are interested in weighing inspection requirements (i.e., percent of modules rejected and subjected to detailed inspection) against the quality that is achieved, for various BDFs. We estimate inspection requirements by noting that all modules in the *REJECT* column of Table 2 must be inspected; this is the count $C_{12}+C_{22}$. Thus, the proportion of modules that must be inspected is given by:

$$I = (C_{12} + C_{22})/n$$ (10)

For the example, I=((344+541)/1397)*100=63.35% and the percentage accepted is 1-I = 36.65%.

3.2.6. Summary of Validation Results

Table 3 summarizes the results of the validation example. The properties of *dominance* and *concordance* are evident in these validation results and in other data we have analyzed from the *Space Shuttle*. That is, a point is reached in adding metrics where *Discriminative Power* is not increased because: 1) the contribution of the dominant metrics in correctly classifying quality has already taken effect and 2) additional metrics essentially replicate the classification results of the dominant metrics -- the *concordance* effect. This result is due to the property of the BDF used as an OR function, causing a module to be rejected if only one of its metrics exceeds its critical value.

3.3. Stage 3: Apply a Stopping Rule for Adding Metrics

It is important to strike a balance between quality and cost (i.e., between RFP and I). Thus we add metrics until the ratio of the relative change in RFP to the relative change in I is maximum, as given by the *Quality Inspection Ratio* in equation (11), where i refers to the previous RFP and I:

$$QIR = (|\Delta RFP|/RFP_i)/(\Delta I/I_i)$$ (11)

For the example, QIR(P,S--P,S, E2)= ((|.2.17-2.95|)/2.95)/((63.35-60.13)/60.13)=4.91. Therefore, we stop adding metrics after *eta2* (E2) has been added.

3.3.1. Comparison of BDF Validation with Application Results

In order to compare validation with application results, we first show how BDF Table looks in the Design phase of Build 2 in Figure 1, when only the metrics M_{ij} and their critical values MC_j are available. This is shown in Table 4, where the "?" indicates that the quality factor data F_i are not available when the validated metrics are used in the quality control function of Build 2. During the Design phase of Build 2, modules are classified according

to the criteria that have been described. Whereas 36.65% (512/1397) and 63.35% (885/1397) modules were accepted and rejected, respectively, during Build 1 (see Table 2), 26.95% (228/846) and 73.05 % (618/846) modules were accepted and rejected, respectively, during Build 2 (see Table 4). The rejected modules would be given priority attention (i.e., subjected to rigorous inspection).

A comparison of the Validation (Build 1) with the Application (Build 2) with respect to statistical and application criteria are shown in Table 5. To have a basis for comparison with the validation results, we computed the values shown in Table 5 retrospectively (i.e., after Build 2 was far enough along to be able to collect all of the quality factor data at the conclusion of the Test phase). The values for Build 2 are the actual quality delivered to maintenance, as shown during the Test phase of Figure 1. The results of the two builds are comparable. Note that the same critical values computed during Build 1 were used on Build 2. This procedure is necessary because the quality factor data that is used in the K-S test in Stage 1 is not available during the Design Phase of Build 2 in Figure 1. This transferability of model parameters is key to our process because the point of validation is to apply its results to other but similar software when the quality factor data is not available for the latter. Also, we have found that to apply this approach, Build 2 does not have to be a direct descendant of Build 1. Builds 1 and 2 do not have this relationship.

3.4. Stage 4: Form a Set of Relative Critical Value Metrics (RCVD)

Granularity of data is an issue that does not seem to have been discussed much in the literature but one that we have found to be of great importance in metrics analysis. By granularity we refer to the level of data (e.g., module, module sets, build) that will yield useful results when the data are used in a model. This was an issue in our research to develop an RCVD suitable for use as a second level discriminant in controlling and predicting quality. By second level we mean that the RCVD comes into play after the optimal BDF has done its job of either accepting or rejecting a module. Although the BDF is very useful, it does not indicate the *degree* of quality (e.g., number of DRs) on a rejected module or set of rejected modules. Our original objective was to provide discrimination at the module level (i.e., rank the *drcount* in modules by RCVD). Due to the large number of modules with zero DRs (58.77% and 50.59% for Build 1 and Build 2, respectively) and the large variability of the data, this did not prove feasible. However, by sorting the modules by RCVD and finding its percentile distribution and the corresponding *drcount* percentile distribution, we were able to identify key points in the plots of these distributions. We call these points *break points*. These are points in the percentile distributions where the slope of the percentile curve starts to increase sharply. An example is shown in

Figure 3, where percentile *drcount* is plotted against percentile *prologue size*. A break point occurs at .80 percentile (80%) on the X-axis. This corresponds to RCVD (*prologue size*)=0.517. This value corresponds to a Y-axis value of .35 (35%). Thus for values of RCVD greater than .0517, we estimate that the RCVD would identify 65% of the *drcount*. Thus we see that a difference of only .20 percentile (1.00-.80) of the RCVD accounts for a difference in .65 percentile (1.00-.35) of the *drcount*. In order to implement this process, we validate function (12) for sets of metrics during the Test Phase of Build 2, in Figure 1, when the quality factor data F_i are available. Then we apply function (12) during the Design Phase of Build 2, when no quality factor date is available for Build 2.

$$\vee \left(M_{ij} > MC_j \right) \wedge RCVD_{ij} \qquad (12)$$

This means that in addition to rejecting modules -- the function performed by the BDF -- there is further classification performed by the RCVD. Any modules that evaluate to *true* in (12), would receive special attention because the likelihood is that they would contain multiple DRs. This is illustrated in Table 6 where 65.37% of the *drcount* is identified by RCVD (*prologue size*) in combination with the BDF on Build 1, corresponding to a *drcount* density of 6.08. This is in contrast with a density of .80 on modules where (12) does not evaluate to *true* and 2.85 when the BDF alone is used. Similar results are observed for Build 2 in Table 6. These results indicate the quality that would be delivered to maintenance unless action is taken in inspection and test to correct the defects.

We experimented with using all six metrics of Table 1 in the RCVD. We used all six in order to have sufficient data to make the computation feasible. $\overline{RCVD}$ was worse than RCVD (*prologue size*), as can be seen in Table 6, in terms of both percentage of *drcount* classified and *drcount* density. Since RCVD (*prologue size*) is much easier to compute, it was the preferred RCVD to apply to Build 2, as shown in Table 6. This result is due to the *dominance* and *concordance* properties of metrics mentioned earlier. In addition, the result is due to the fact that *prologue size* contains a thorough change history comprised of the following notations in the program listing: module; purpose of the module; specification reference; change request; discrepancy report; release; release date; revision level; programmer; description of change; listing of statements affected by the change; indication of whether a statement is added, deleted, or changed; and program comments. We use *prologue size* as a predictor of *drcount* in the *aggregate* (i.e., the cumulative quantity of entries in the program), not on a one-for-one basis of a change possibly resulting in a DR.

A seemingly trivial but yet important aspect of this stage of the analysis was demonstrating the usefulness of sorting data to examine their distributions and the flexibility for doing this provided by a spreadsheet program.

3.5. Stage 5: Identify Quality Limit Predictors

The final stage of the analysis involves identifying regression equations for predicting the average and limits of quality (e.g., *drcount*) of module sets, $F_i = f(M_{ij})$, during the Test Phase of Build 1, as shown in Figure 1. This process is desirable because BDFs and RCVDs are not capable of predicting quality limits. During the Test phase of Build 1, regression coefficients are estimated and the resultant equation is applied, during the Design Phase of Build 2, to predict the quality limits that would be delivered to maintenance unless action is taken to correct the defects. As in the case of forming the RCVDs, granularity of data was an issue. Again, because of the large number of modules with zero *drcount* and the large variability of the data, prediction at the individual module level was not feasible. However, applying our earlier regression work for the *Space Shuttle* [13], where we found that if we divided the data into the appropriate number of frequency classes (i.e., modules sets), according to Sturges' rule [14], usable regression equations could be developed based on the averages computed for the classes. In that work, we only predicted average values. We now extend the approach to include predicting quality limits. We experimented with various sets of predictor variables. The model results are shown in Table 7. The equation we selected is the exponential function using average statements (ave S):

$$avedrcount = \exp(0.1137 + 0.0056697 * aveS) \quad (13)$$

This equation was selected for application to Build 2 for the following reasons: 1) lowest Mean Square Error (MSE) in Table 7; 2) fair accuracy in predicting Build 1 *drcount*; 3) theoretical consideration that the rate of change of *drcount* with module size would vary with module size (property of exponential distribution); and the relative ease of collecting size data. Although the F-ratio and R^2 are impressive for the linear function using *nodes*, this equation has a relatively high MSE and the collection of *nodes* requires the use of a metrics analyzer.

Prediction results are shown in Figures 4 -- 7. The figures show the following for average *drcount* for sets of 100 modules (1 -- 100, 101 -- 200, etc.): Figure 4, actual and predicted values for Build 1; Figure 5, actual and predicted limits for Build 1; Figure 6, actual and predicted values for Build 2; and Figure 7, actual and predicted limits for Build 2. Figure 7 shows that the prediction limits bracket the actual values for Build 2. This is another example of retrospective analysis: once the quality factor data F_i are available during the Test Phase of Build 2, Figure 1, the actual *drcount* can be compared with the predictions. In the application of the prediction equation, the software manager would compute the average size of sets of modules and predict the *drcount* and the limits of *drcount* for each module set, as shown in Figures 6 and 7, respectively.

4. Summary and Conclusions

We developed a new metric, Relative Critical Value Deviation (RCVD), for classifying and predicting software quality. When the granularity of data was considered, the RCVD proved to be a useful indicator of the degree to which software quality deviates from a specified norm. We discovered that the major application of the RCVD was to prioritize the effort required to achieve quality goals. At the outset we posed several questions that the software manager wants answered concerning software quality. We provided an integrated set of models based on Boolean discriminant functions, RCVDs, and regression equations to address these questions. We made a thorough evaluation of two builds - one was used for validation and the other for application -- using a five-stage analysis approach. In the three areas of our modeling effort, the predictions for the application build were close to the actual values. Based on these preliminary results and the fact that we have done analysis on additional *Space Shuttle* data, we feel that the *models*, not the specific numerical results, are transferable to other organizations, if the models are applied within and not across application domains. However, to increase our confidence in the results, in future research we will examine several additional builds of the *Space Shuttle* flight software. Finally, we found that mundane aspects of the analysis like data sorting to discover information about distributions of data and the use of spreadsheet calculations significantly aided the analysis.

5. Acknowledgments

6. References

[1] Norman F. Schneidewind, " A Software Metrics Model for Quality Control", Proceedings of the International Metrics Symposium, Albuquerque, New Mexico, November 7, 1997, pp. 127-136.

[2] Norman F. Schneidewind, "A Software Metrics Model for Integrating Quality Control and Prediction", Proceedings of the International Symposium on Software Reliability Engineering, Albuquerque, New Mexico, November 4, 1997, pp. 402-415.

[3]   A. A. Porter and R. W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees", IEEE Software, Vol. 7, No. 2, March 1990, pp. 46-54.

[4]   Lionel C. Briand, John Daly, Victor Porter, and Jürgen Wüst, "Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems", Proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, November 4-7, 1998, pp. 334-343.

[5]   Taghi M. Khoshgoftaar and Edward B. Allen. "Logistic Regression Modeling of Software Quality", Department of Computer Science & Engineering, Florida Atlantic University, TR-CSE-97-24, March, 1997.

[6]   Taghi M. Khoshgoftaar, Edward B. Allen, Robert Halstead, and Gary P. Trio, "Detection of Fault-Prone Software Modules During a Spiral Life Cycle", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 69-76.

[7]   Taghi. M. Khoshgoftaar, Edward B. Allen, Kalai Kalaichelvan, and Nishith Goel, "Early Quality Prediction: A case Study in Telecommunications", IEEE Software, Vol. 13, No. 1, January 1996, pp. 65-71.

[8]   Taghi M. Khoshgoftaar and Edward B. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software", Proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, November 4-7, 1998, pp. 344-353.

[9]   Niclas Ohlsson and Hans Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches", IEEE Transactions on Software Engineering, Vol. 22, No. 12, December 1996, pp. 886-894.

[10]   Standard for a Software Quality Metrics Methodology, Revision, IEEE Std 1061-1998, 31 December, 1998.

[11]   Norman F. Schneidewind, "Methodology for Validating Software Metrics", IEEE Transactions on Software Engineering, Vol. 18, No. 5, May 1992, pp. 410-422.

[12]   Allen P. Nikora and John C. Munson, "Determining Fault Insertion Rates for Evolving Software Systems", Proceedings of the Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, November 4-7, 1998, pp. 306-315.

[13]   Norman F. Schneidewind, "Software Metrics Validation: Space Shuttle Flight Software Example", Annals of Software Engineering, J. C. Baltzer AG, Science Publishers, 1(1995)287-309.

[14]   J. D. Jobson, Applied Multivariate Data Analysis, Volume II:, Springer-Verlag, 1992.

Table 1: Kolmogorov-Smirnov Distance for *drcount*=0 vs. *drcount*>0 Validation: Build 1 (n=1397 modules)

Metric (symbol)	Definition (counts per module)	Critical Value	Distance	α	Rank
prologue size (P)	change history line count in module listing	63	0.592	0.005	1
statements (S)	executable statement count	27	0.505	0.005	2
eta2 (E2)	unique operand count	45	0.472	0.005	3
loc (L)	non-commented lines of code count	29	0.462	0.005	4
eta1 (E1)	unique operator count	9	0.430	0.005	5
nodes (N)	node count (in control graph)	17	0.427	0.005	6

Table 2: Boolean Discriminant Function: Validation (Build 1)

	$\wedge(M_{ij} \le MC_j)$ $P_i \le 63 \wedge S_i \le 27 \wedge E2_i \le 45$	$\vee(M_{ij} > MC_j)$ $P_i > 63 \vee S_i > 27 \vee E2_i > 45$	
High Quality $F_i \le FC$ *drcount*=0	$C_{11}=477$	$C_{12}=344$ Type 2	$n_1=821$
Low Quality $F_i > FC$ *drcount*>0	$C_{21}=35$ Type 1	$C_{22}=541$	$n_2=576$
	$N_1=512$ RF=56	$N_2=885$	n=1397 TF=2579
	ACCEPT	REJECT	

Table 3: *Discriminative Power* Validity Evaluation (Build 1, n=1397 modules)

Metric Set	Critical Values				Statistical Criteria		Application Criteria			
	P	S	E2	L	P_1 %	P_2 %	LQC %	RFP %	QIR	I %
P	63				6.23	15.10	84.90	6.13	-	50.11
P, S	63	27			3.22	22.12	92.19	2.95	2.59	60.13
P, S, E2	63	27	45		2.51	24.62	93.92	2.17	**4.91**	63.35
P, S, E2, L	63	27	45	29	2.00	29.35	95.14	1.78	2.16	68.58
K-S Distance	0.592	0.505	0.472	0.462						

P: prologue size, S: statements, E2: eta2, L: lines of code

Table 4: Boolean Discriminant Function: Application (Build 2)

	$\wedge(M_{ij} \le MC_j)$ $P_i \le 63 \wedge S_i \le 27 \wedge E2_i \le 45$	$\vee(M_{ij} > MC_j)$ $P_i > 63 \vee S_i > 27 \vee E2_i > 45$	
High Quality ?	?	Type 2 ?	?
Low Quality ?	Type 1 ?	?	?
	$N_1=228$	$N_2=618$	n=846
	ACCEPT	REJECT	

Table 5: Comparison of Validation (Build 1, n=1397 modules) with Application (Build 2, n=846 modules)

Metric Set	Critical Values			Statistical Criteria		Application Criteria			
	P	S	E2	P₁ %	P₂ %	LQC %	RFP %	QIR	I %
Validation P, S, E2	63	27	45	2.51	24.62	93.92	2.17	4.91	63.35
Application P, S, E2	63	27	45	3.07	26.71	93.78	2.69	9.11	73.05

P: prologue size, S: statements, E2: eta2

Table 6: Comparison of Relative Critical Value Deviation (RCVD) *Discriminative Power*

	Build 1 (Validation)		Build 2 (Application)
	$\overline{RCVD}$ (six metrics)	RCVD (*prologue size*)	RCVD (*prologue size*)
.80 Percentile RCVD Value (Break Point)	.1026	.0517	.0777
BDF ∧ RCVD	((P>63)∨(S>27)∨(E2>45)) ∧(RCVD>.1026)	((P>63)∨(S>27)∨(E2>45)) ∧(RCVD>.0517)	((P>63)∨(S>27)∨(E2>45)) ∧(RCVD>.0777)
drcount identified (percent)	1400 (54.28)	1686 (65.37)	1002 (62.74)
modules with *drcount* identified (percent)	263 (18.83)	280 (20.04)	173 (20.45)
drcount density (*drcount*/module)	5.32	6.02	5.79
drcount density for other modules	1.04	.80	.88
BDF	((P>63)∨(S>27)∨(E2>45))		
drcount density	2.85		2.51

1. $\overline{RCVD}$ (six metrics): mean of RCVDs of six metrics in Table 1
2. *drcount* identified: count of DRs on modules rejected by BDF ∧ RCVD; percent of total DRs
3. modules with *drcount* identified: count of modules rejected by BDF ∧ RCVD; percent of total modules
4. *drcount* density: *drcount*/module count
5. *drcount density* for other modules: modules other than those rejected by BDF ∧ RCVD

Table 7: Regression Equation Summary for Predicting ave*drcount*

Predictor Variables	Type	F	R²	MSE	Mean Residual	Predicted Build *drcount*	Actual Build *drcount*
				Build 1: Validation			
aveS	Exponential	56.94	.851	0.702	.0000	2377	2579
aveN	Linear	283.13	.966	1.545	.0000	2241	2579
aveS, aveN	Exponential	39.84	.899	0.754	.0000	2404	2579
				Build 2: Application			
aveS	Exponential	56.94	.851	0.437		1637	1597

S: statements, N: nodes, MSE: mean square error computed between predicted and actual *drcount*

Development →

Build 1: Validation   Build 2: Application   Maintenance of Build 2

Design	Test

Design	Test

MCj → MCj

M_{ij}   F_i   M_{ij} → Control & Predict Quality →

RCVD_{ij}   RCVD_{ij}

F_i=f(M_{ij})   F_i=f(M_{ij})   F_i: Known Quality →

M_{ij} : Metric j on Module i   RCVD_{ij} : Relative Critical Value Deviation for Metric j on Module i
MC_j : Metric j Critical Value
F_i : Quality Factor on Module i   F_i=f(M_{ij}): Quality Limits Predictor
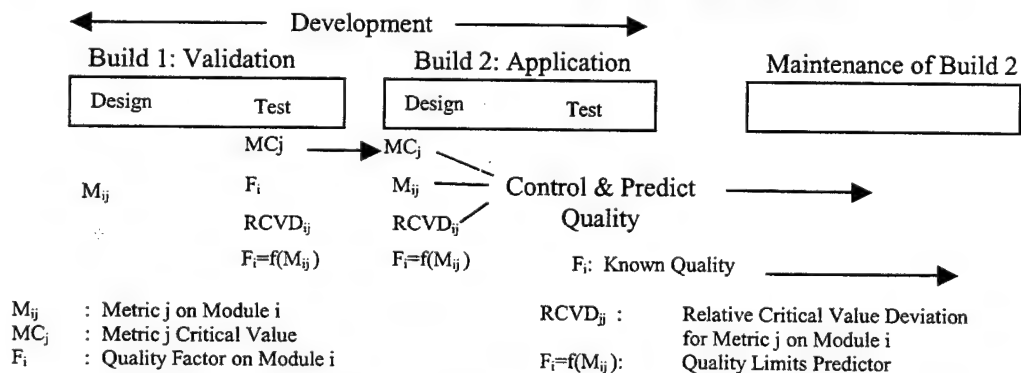
Figure 1. Measurement Process
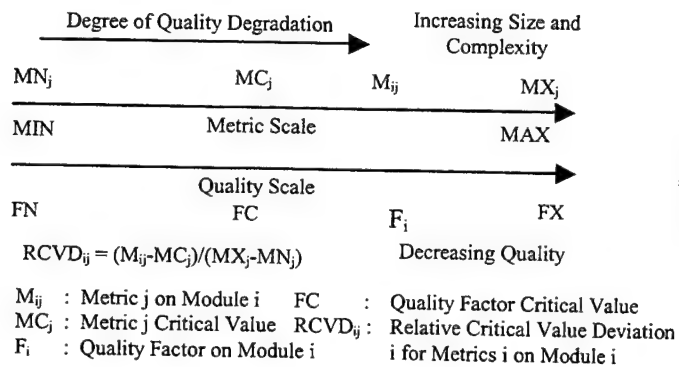
Figure 2. Quality Thermometer
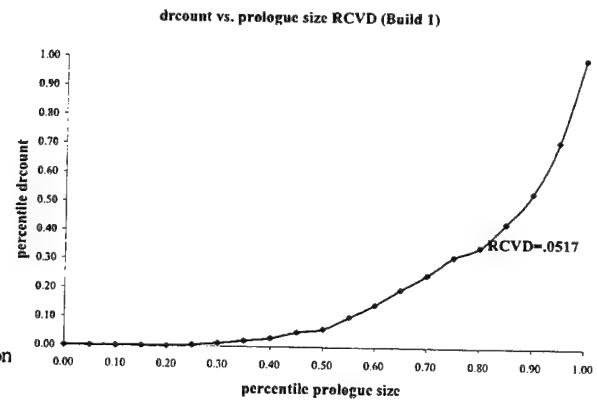
Figure 3. drcount and prologue size RCVD percentiles
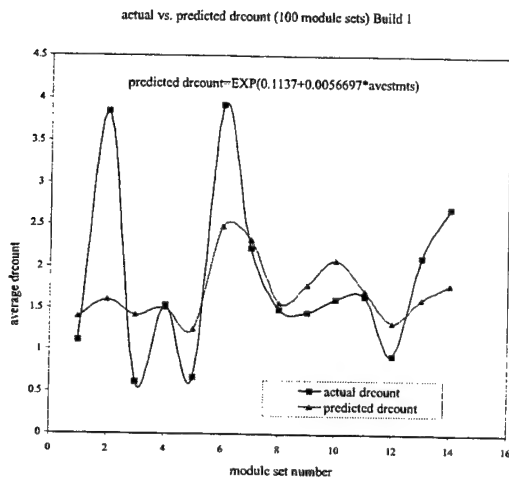
Figure 4. Predicted vs. Actual drcount (Build 1)   Figure 5. Predicted Limits vs. Actual drcount (Build 1)
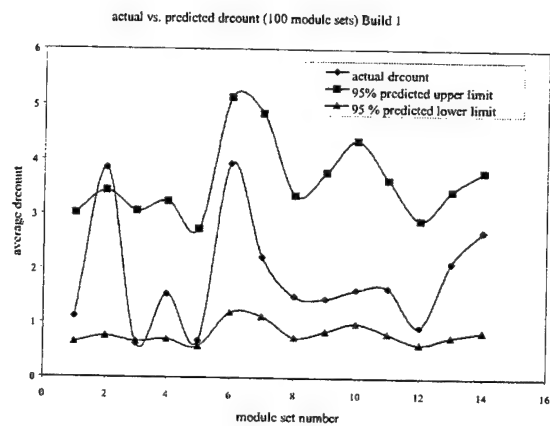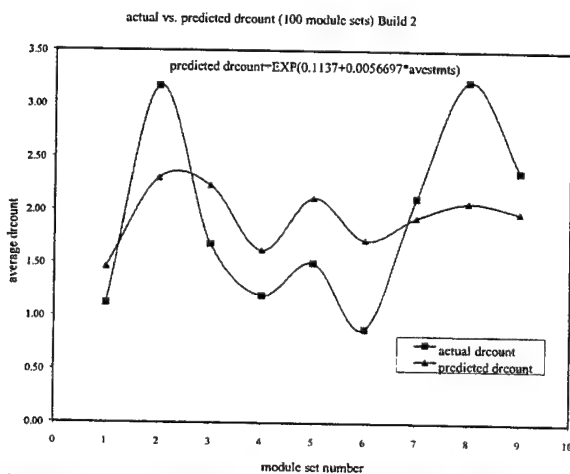
Figure 6. Predicted vs. Actual drcount (Build 2)   Figure 7. Predicted Limits vs. Actual drcount
(Build 2)
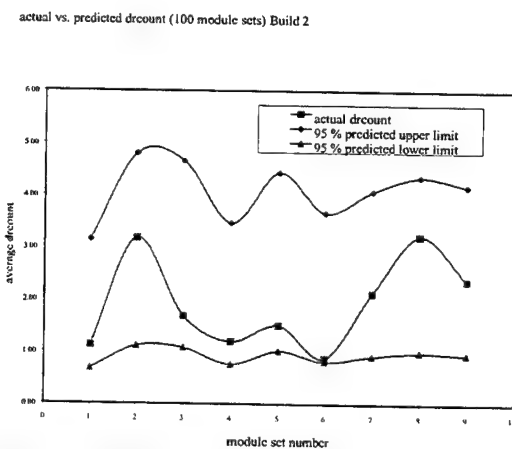
Investigation of the Risk to Software Reliability of Requirements Changes
The 1999 NASA Workshop on Risk Management, Morgantown, West Virginia, October 28-29, 1999, 13 pages.

Norman F. Schneidewind

BACKGROUND

While software design and code metrics have enjoyed some success as predictors of software quality attributes such as reliability [KHO961, KHO962, LAN95, MUN96, OHL96], the measurement field is stuck at this level of achievement. If measurement is to advance to a higher level, we must shift our attention to the front-end of the development process, because it is during system conceptualization that errors in specifying requirements are inserted into the process. A requirements change may induce ambiguity and uncertainty in the development process that cause errors in implementing the changes. Subsequently, these errors propagate through later phases of development and maintenance. These errors may result in significant risks associated with implementing the requirements. For example, reliability risk (i.e., risk of faults and failures induced by changes in requirements) may be incurred by deficiencies in the process (e.g., lack of precision in requirements). Although requirements may be specified correctly in terms of meeting user expectations, there could be significant risks associated with their implementation. For example, correctly implementing user requirements could lead to excessive system size and complexity with adverse effects on reliability or there could be a demand for project resources that exceeds the available funds, time, and personnel skills. Interestingly, there has been considerable discussion of project risk (e.g., the consequences of cost overrun and schedule slippage) in the literature [BOH91] but not a corresponding attention to reliability risk.

Risk in the Webster's New Universal Unabridged Dictionary is defined as: "the chance of injury; damage, or loss" [WEB79]. Some authors have extended the dictionary definition as follows: "Risk Exposure=Probability of an Unsatisfactory Outcome*Loss if the Outcome is Unsatisfactory" [BOH91]. Such a definition is frequently applied to the risks in managing software projects such as budget and schedule slippage. In contrast, our application of the dictionary definition pertains to the risk of executing the software of a system where there is the chance of injury (e.g., crew injury or fatality), damage (e.g., destruction of the vehicle), or loss (e.g., loss of the mission) if a serious software failure occurs during a mission. We use risk factors to indicate the degree of risk associated with such an occurrence.

The generation of requirements is not a one-time activity. Indeed, changes to requirements can occur during maintenance. When new software is developed or existing software is changed in response to new and changed requirements, respectively, there is the potential to incur reliability risks. Therefore, in assessing the effects of requirements on reliability, we should deal with *changes* in requirements throughout the life cycle.

In addition to the relationship between requirements and reliability, there are the intermediate relationships between requirements and complexity and between complexity and reliability. These relationships may interact to put the reliability of the software at risk because the requirements changes may result in increases in the size and complexity of the software that may adversely affect reliability.

OBJECTIVES AND EXPECTED SIGNIFICANCE

Objectives

Given the lack of emphasis in metrics research on the critical role that requirements play in determining reliability, combined with our experience and interest in metrics and reliability, we are motivated to investigate the following issues:

- What is the relationship between requirements attributes and reliability? That is, are there requirements attributes that are strongly related to the occurrence of defects and failures in the software?

- What is the relationship between requirements attributes and software attributes like complexity and size? That is, are there requirements attributes that are strongly related to the complexity and size of software?

- Is it feasible to use requirements attributes as predictors of reliability? That is, can *static* requirements change attributes like the size of the change be used to predict reliability in *execution* (e.g., time to next failure, number of failures)?

- Are there requirements attributes that can discriminate between high and low reliability, thus qualifying these attributes as predictors of reliability?

- Which requirements attributes pose the greatest risk to reliability?

An additional objective is to develop a framework that other researchers could use for the following: 1) analyze the relationships between requirements changes, complexity, and reliability, and 2) assess and predict reliability risk as a function of requirements changes.

Significance

This research is significant because the field of software engineering lacks the capability to quantitatively assess and predict the effect of a requirements change on the reliability of the software. Much of the research and literature in software metrics concerns the measurement of code characteristics [NIK98]. This is satisfactory for evaluating product quality and process effectiveness once the code is written. However, if organizations use measurement plans that are limited to measuring code, they will be deficient in the following ways: incomplete, lack coverage (e.g., no requirements analysis and design), and start too late in the process. For a measurement plan to be effective, it must start with requirements and continue through to

operation and maintenance. Since requirements characteristics directly affect code characteristics and hence reliability, it is important to assess their impact on reliability when requirements are specified.

RESEARCH PLAN

Our research is aimed at conducting experiments to see whether it is feasible to develop a mapping between changes in requirements to changes in software complexity and reliability. In other words, we will investigate whether the following implications hold, where R represents requirements, C represents complexity, and F represents failure occurrence (i.e., reliability): $\Delta R \Rightarrow \Delta C \Rightarrow \Delta F$. We include changes in size and documentation in changes in complexity.

By retrospectively analyzing the relationship between requirements and reliability, we will be able to construct models that can predict reliability as a function of requirements changes. In order to quantify the effect of a requirements change, we use various risk factors that are defined as the attribute of a requirement change that can induce reliability or project risk. Various examples of risk factors are shown in the section *Risk Factors*. We propose to statistically analyze specified risk factors to see in what way, if any, they are associated with reliability. In particular, we want to identify those factors that have an adverse effect on reliability. In addition to risk factors, we can also use the number of requirements change requests on modules (see Table 4, *Data Sources* section). The number and rate of occurrence of these requests will be considered as additional potential risk factors.

Experiments

Several experiments will be conducted to see whether $\Delta R \Rightarrow \Delta C \Rightarrow \Delta F$ is confirmed for specified data sets. These include discriminant analysis, trend analysis, and reliability prediction. Examples of the data that would be used are shown in the *Data Sources* section.

Discriminant Analysis

Using the null hypothesis, H_o: A given set of risk factors is *not* a discriminator of reliability versus the alternate hypothesis H_1: A given set of risk factors is a discriminator of reliability, we will use categorical data analysis and discriminant analysis to test the hypothesis. A similar hypothesis will be used to assess whether factors can serve as discriminators of complexity. We will use the rich set of requirements, requirements risk factors, reliability, and metrics data we have from the Space Shuttle to test our hypotheses. We will develop a discriminate function comprised of a linear or Boolean function of the risk factors. We will evaluate this function to see whether it can accurately discriminate between those requirements changes that caused a one or more failures and those that did not. We will draw on our experience in applying categorical data analysis and discriminant analysis for classifying the quality of software based on using Boolean discriminant functions that are comprised of both a set of metrics and corresponding critical values [SCH971].

To minimize the effects of a large number of variables that interact in some cases, a statistical categorical data analysis will be performed incrementally. We will use only one category of risk factor at a time to see the effect of adding an additional discriminator on the ability to correctly classify modules that have discrepancy reports (i.e., a report that documents deviations between specified and observed software behavior) and those that do not.

Trend Analysis

In our work on analyzing the relationship between product reliability and process stability, we developed a generalized relative Change Metric (CM) that represents trend information (e.g., changes in reliability across releases) in a single metric [SCH98]. CM is independent of the scales of the measured quantities. Although looking for a trend on a graph is useful, it is not a precise way of measuring and comparing trends, particularly if the graph has peaks and valleys and the measurements are made at discrete points in time. We will use this metric to measure changes in risk factors, complexity, and reliability across releases or builds of the software and compare them to see whether trends are as expected (i.e., increases in risk factors are accompanied by increases in complexity and decreases in reliability). The following is an example of computing CM for the reliability metric *failures/KLOC*:

1. Note the change in a metric from one release to the next (i.e., release j to release j+1).

2.a. If the change is in the desirable direction (e.g., Failures/KLOC decrease), treat the change in 1 as positive.

b. If the change is in the undesirable direction (e.g., Failures/KLOC increase), treat the change in 1 as negative.

3. a. If the change in 1 is an increase, divide it by the value of the metric in release j+1.
   b. If the change in 1 is a decrease, divide it by the value of the metric in release j.

4. Compute the average of the values obtained in 3, taking into account sign. This is the change metric (CM). The CM is a quantity in the range −1, 1. A positive value indicates a favorable trend; a negative value indicates an unfavorable trend. The numeric value of CM indicates the degree of stability or instability. The standard deviation of these values can also be computed. The average of the CM for a set of metrics can be computed to obtain an overall change metric. An example of calculating CM for MTTF and Failures/KLOC is shown in Table 1 for various Operational Increments (releases) of Shuttle software, where an Operational Increment (OI) is a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements. Figure 1 shows the corresponding plot of Failures/KLOC across the releases.

Table 1: Example Computations of Change Metric (CM)				
Operational Increment	MTTF (Days)	Relative Change	Failures/KLOC	Relative Change
A	179.7		0.750	
B	409.6	0.562	0.877	-0.145
C	406.0	-0.007	1.695	-0.483
D	192.3	-0.527	0.984	0.419
E	374.6	0.487	0.568	0.423
J	73.6	-0.805	0.238	0.581
O	68.8	-0.068	0.330	-0.272
	CM	**-0.060**	**CM**	**0.087**

Reliability Prediction

If the discriminant analysis and trend analysis prove successful with respect to discriminating reliability as a function of risk factors, we will use the results as scale factors on reliability predictions. An example of this approach is shown in Figure 2, which is a plot of failure rate versus test time for one of the Shuttle OIs. This plot shows a decreasing trend, after an initial period of instability (i.e., increasing rate as personnel learn how to maintain new software). Figure 2 shows that stability is achieved (i.e., failure rate asymptotically approaches zero with increasing test time).There are two types of risks indicated on the diagram. One is the reduction in risk with increased testing. We have done research work on this type of risk [KEL95]. The other risk – a subject of this proposal – is represented by a family of failure rate curves (two shown). The concept is that the lower curve (lower failure rate) would be associated with requirements changes that have lower reliability risk than the upper curve. Thus, we would predict on an ordinal scale that for a given amount of test time. requirements changes that have lower risk would result in higher reliability (lower failure rate). Alternatively, for a given failure rate, we would predict that requirements that have lower reliability risk would require less test time to achieve the specified reliability goal.

RISK FACTORS

One of the software maintenance problems of the NASA Space Shuttle Flight Software organization is to evaluate the risk of implementing requirements changes. These changes can affect the reliability and maintainability of the software. To assess the risk of change, the software development contractor uses a number of risk factors. which are described below. The risk factors were identified by agreement between NASA and the development contractor based on assumptions about the risk involved in making changes to the software. This formal process is called a risk assessment. No requirements change is approved by the change control board without an accompanying risk assessment. During risk assessment, the development contractor will attempt to answer such questions as: "Is this change highly complex relative to other software changes that have been made on the Shuttle?" If this were the case, a high-risk value

would be assigned for the complexity criterion. To date this *qualitative* risk assessment has proven useful for identifying possible risky requirements changes or, conversely, providing assurance that there are no unacceptable risks in making a change. However, there has been no *quantitative* evaluation to determine whether, for example, high risk factor software was really less reliable and maintainable than low risk factor software. In addition, there is no model for predicting the reliability and maintainability of the software, if the change is implemented. Our research will address both of these issues.

We had considered using requirements attributes like completeness, consistency, correctness, etc as risk factors [DAV90]. While these are useful generic concepts, they are difficult to quantify. Although some of the following risk factors also have qualitative values assigned, there are a number of quantitative factors, and many of the factors deal with the execution behavior of the software (i.e., reliability), which is our research interest.

The following are the definitions of the risk factors, where we have placed the factors into categories and have provided our interpretation of the question the factor is designed to answer. In addition, we added the risk factor *requirements specifications techniques* because we feel that this one could represent the highest reliability risk of all the factors if a technique leads to misunderstanding of the intent of the requirements.

Shuttle Flight Software Requirements Change Risk Factors

If the answer to a yes/no question is "yes", it means this is a high-risk change with respect to the given factor. If the answer to a question that requires an estimate is an anomalous value, it means this is a high-risk change with respect to the given factor.

Complexity Factors

o   Qualitative assessment of complexity of change (e.g., very complex)

   - Is this change highly complex relative to other software changes that have been made on the Shuttle?

o Number of modifications or iterations on the proposed change
   - How many times must the change be modified or presented to the Change Control Board (CCB) before it is approved?

Size Factors

o   Number of lines of code affected by the change
   - How many lines of code must be changed to implement the change?

o   Size of data and code areas affected by the change
   - How many bytes of existing data and code are affected by the change?

Criticality of Change Factors

o Whether the software change is on a nominal or off-nominal program path (i.e., exception condition)
  - Will a change to an off-nominal program path affect the reliability of the software?

o Operational phases affected (e.g., ascent, orbit, and landing)
  - Will a change to a critical phase of the mission (e.g., ascent and landing) affect the reliability of the software?

Locality of Change Factors

o The area of the program affected (i.e., critical area such as code for a mission abort sequence)
  - Will the change affect an area of the code that is critical to mission success?

o Recent changes to the code in the area affected by the requirements change
  - Will successive changes to the code in one area lead to non-maintainable code?

o New or existing code that is affected
  - Will a change to new code (i.e.. a change on top of a change) lead to non-maintainable code?

o Number of system or hardware failures that would have to occur before the code that implements the requirement would be executed
  - Will the change be on a path where only a small number of system or hardware failures would have to occur before the changed code is executed ?

Requirements Issues and Function Factors

o Number and types of other requirements affected by the given requirement change (requirements issues)
  - Are there other requirements that are going to be affected by this change? If so, these requirements will have to be resolved before implementing the given requirement.

o Possible conflicts among requirements changes (requirements issues)
  - Will this change conflict with other requirements changes (e.g., lead to conflicting operational scenarios)

o Number of principal software functions affected by the change
  - How many major software functions will have to be changed to make the given change?

Performance Factors

o Amount of memory required to implement the change

207

- Will the change use memory to the extent that other functions will be not have sufficient memory to operate effectively?

. o Effect on CPU performance
   - Will the change use CPU cycles to the extent that other functions will not have sufficient CPU capacity to operate effectively?

Personnel Resources Factors

o Number of inspections required to approve the change.

o Manpower requirements required to implement the change
   - Will the manpower required to implement the software change be significant?

o Manpower required to verify and validate the correctness of the change
   - Will the manpower required to verify and validate the software change be significant?

Tools Factors

o Any software tools creation or modification required to implement the change
   - Will the implementation of the change require the development and testing of new tools?

o Requirements specifications techniques (e.g., flow diagram, state chart, pseudo code, control diagram).
   - Will the requirements specification method be difficult to understand and translate into code?

DATA SOURCES

We have access to several sets of data from the Space Shuttle of the following types:

- Pre-release and post release failure data from the Space Shuttle from 1983 to the present. An example of post-release failure data is shown in Table 2 (data provided by US Alliance, Houston, Texas).

Table 2

Failure Found On Operational Increment	Days from Release When Failure Occurred	Discrepancy Report #	Severity	Failure Date	Release Date	Module in Error
Q	75	110402	2	05-19-97	03-05-97	10

- Risk factors for the Shuttle *Three Engine Out Auto Contingency* and *Single Global Positioning System* software. This software was released to NASA by the developer on 10/18/95 and 3/5/97, respectively. An example of a partial set of risk factor data is shown in Table 3 (data provided by US Alliance, Houston, Texas).

Table 3

Change Request Number	SLOC Changed	Complexity Rating of Change	Criticality of Change	Number of Principal Functions Affected	Number of Modifications Of Change Request	Number of Requirements Issues	Number of Inspections Required	Manpower Required to Make Change
107734	1933	4	3	27	7	238	12	209.3 MW

- Metrics data for 1400 Shuttle modules, each with 26 metrics. An example of a partial set of metric data is shown in Table 4 (data provided by Prof. John Munson, University of Idaho).

Table 4

Module	Operator Count	Operand Count	Statement Count	Path Count	Cycle Count	Discrepancy Report Count	Change Request Count
10	3895	1957	606	998	4	14	16

We will use the Shuttle data to test our hypothesis about the ability of risk factors to discriminate between levels of reliability and complexity and the Jet Propulsion Laboratory X-2000 system – the latest planetary vehicle. This project provides a rare opportunity to work with the software development team and testers to establish a measurement plan form the inception of a project as opposed to the usual situation of having to intervene in an on-going project. We plan to instrument the software system for obtaining measurements throughout the development and maintenance process.

LONG-TERM GOALS

This research is another in the series of our software measurement projects that has included software reliability modeling and prediction, metrics analysis, risk analysis, and maintenance stability analysis [SCH98]. We have been involved in the development and application of software reliability models for many years [SCH93, SCH92]. Our models, as is the case in general in software reliability, use failure data as the driver. This approach has the advantage of using a metric that represents the dynamic behavior of the software. However, this data is not available until the test phase. Predictions at this phase are useful but it would be much more useful to predict at an earlier phase – preferably during requirements analysis—when the cost of error correction is relatively low. Thus, there is great interest in the software reliability and metrics field in using static attributes of software in reliability modeling and prediction.

Integrating Risk Analysis with Reliability Prediction

In the past, we have coupled reliability prediction with risk analysis, but the risk metrics we developed pertained to the risk of the software not meeting remaining failures and time to next failure goals [SCH973]. For example, we have used the *Schneidewind* software reliability model

for integrating reliability and reliability risk predictions for the Space Shuttle. This integration is described below.

Assume that software reliability goals have been specified for a project in terms of *remaining failures* $r(t_t)$ and *time to next failure* $T_F(t_t)$ to be achieved once the software is deployed. Then the criteria for achieving these goals, for a given *test time* t_t, execution or elapsed time, are as follows:

1) predicted remaining failures $r(t_t) < r_c$, where r_c is a specified critical value , and

2) predicted time to next failure $T_F(t_t) > t_m$, where t_m is mission duration.

Remaining Failures Risk Metric

Then we can formulate the normalized *remaining failures risk metric as* follows:

$$(r(t_t)-r_c)/r_c=(r(t_t)/r_c)-1$$

(1)

Equation (1) divides the risk space into three regions, as a function of test time: *positive, zero,* and *negative* values corresponding to $r(t_t) > r_c$, $r(t_t) = r_c$, and $r(t_t) < r_c$, respectively. In terms of risk, these regions correspond to *critical, neutral,* and *desirable,* respectively

Time to Next Failure Risk Metric

Similarly, we can formulate the *time to next failure risk metric* as follows:

$$(t_m-T_F(t_t))/t_m=1-(T_F(t_t))/t_m$$

(2)

Equation (2) divides the risk space into three regions, as a function of test time: *positive, zero,* and *negative* values corresponding to $T_F(t_t) < t_m$, $T_F(t_t) = t_m$, and $T_F(t_t) > t_m$, respectively. In terms of risk, these regions correspond to *critical, neutral,* and *desirable,* respectively.

Based on the premise that no one model suffices for all prediction applications, one of our long-range research goal is to develop an integrated suite of models for various applications. One type of model and predictions were just described. Other members of the suite are our quality metrics and process stability models [SCH971, SCH72, SCH98]. Now we want to change our research emphasis to the prediction of reliability at the earliest possible time in the development process -- to the requirements analysis phase -- that heretofore has been unattainable. We would also like to determine whether there exists a "standard" set of risk factors that could be applied in a variety of applications to reliability prediction. As the X-2000 project matures, and reliability and complexity data become available, we will be able to observe whether the risk factors that we validate on the Shuttle are applicable to the X-2000 for identifying and predicting reliability risk. Also, we may discover additional risk factors on the X-2000 project. Lastly, we will be able to determine whether the numerical results of reliability classification and prediction obtained on the Shuttle scale to the X-2000.

REFERENCES

[BOH91] Barry W. Boehm, "Software Risk Management: Principles and Practices", *IEEE Software*, Vol. 8, No. 1, January 1991, pp. 32-41.

[DAV90] Alan Davis, Software Requirements: Analysis and Specifications, Prentice-Hall. Englewood Cliffs, NJ, 1990.

[KEL95] Ted Keller, Norman F. Schneidewind, and Patti A. Thornton "Predictions for Increasing Confidence in the Reliability of the Space Shuttle Flight Software", *Proceedings of the AIAA Computing in Aerospace 10*, San Antonio, TX, March 28, 1995, pp. 1-8.

[KHO961] Taghi M. Khoshgoftaar, Edward B. Allen, Robert Halstead, and Gary P. Trio, "Detection of Fault-Prone Software Modules During a Spiral Life Cycle", Proceedings of the International Conference on Software Maintenance, Monterey. California, November 4-8, 1996. pp. 69-76.

[KHO962] Taghi. M. Khoshgoftaar, Edward B. Allen. Kalai Kalaichelvan, and Nishith Goel, "Early Quality Prediction: A case Study in Telecommunications, IEEE Software, Vol. 13, No. 1. January 1996, pp. 65-71.

[LAN95] D. Lanning and T. Khoshgoftaar, "The Impact of Software Enhancement on Software Reliability", IEEE Transactions on Reliability, Vol. 44, No. 4, December 1995, pp. 677-682.

[MUN96] John C. Munson and Darrell S. Werries, "Measuring Software Evolution", Proceedings of the Third International Software Metrics Symposium, March 25-26, 1996, Berlin, Germany, pp. 41-51.

[NIK98] Allen P. Nikora, Norman F. Schneidewind, and John C. Munson, IV&V Issues in Achieving High Reliability and Safety in Critical Control Software, Final Report, Volume 1 – Measuring and Evaluating the Software Maintenance Process and Metrics-Based Software Quality Control, Volume 2 – Measuring Defect Insertion Rates and Risk of Exposure to Residual Defects in Evolving Software Systems, and Volume 3 – Appendices, Jet Propulsion Laboratory, National Aeronautics and Space Administration, Pasadena, California, January 19, 1998.

[OHL96] Niclas Ohlsson and Hans Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches", IEEE Transactions on Software Engineering, Vol. 22, No. 12, December 1996, pp. 886-894.

[SCH98] Norman F. Schneidewind, "How to Evaluate Legacy System Maintenance", IEEE Software, Vol. 15, No. 4, July/August 1998, pp. 34-42. Also translated into Japanese and reprinted in: Nikkei Computer Books, Nikkei Business Publications, Inc., 2-1-1 Hirakawacho, Chiyoda-Ku, Tokyo 102 Japan, 1998, pp. 232-240.

[SCH971] Norman F. Schneidewind, "Software Metrics Model for Integrating Quality Control and Prediction≅, Proceedings of the International Symposium on Software Reliability Engineering, Albuquerque, New Mexico, November 4, 1997, pp. 402-415.

[SCH972] Norman F. Schneidewind, "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics", Proceedings of the International Conference on Software Maintenance, Bari, Italy, October 2, 1997, pp. 232-239.

[SCH973] Norman F. Schneidewind, "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, Vol. 46, No.1, March 1997, pp.88-98.

[SCH93] Norman F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data", IEEE Transactions on Software Engineering, Vol. 19. No. 11, November 1993, pp. 1095-1104.

[SCH92] Norman F. Schneidewind and T. W. Keller, "Application of Reliability Models to the Space Shuttle", IEEE Software, Vol. 9. No. 4, July 1992 pp. 28-33.

[WEB79] *Webster's New Universal Unabridged Dictionary*, Second Edition. Simon and Shuster, New York, 1979.

Figure 1. Total Failures per KLOC Across Releases

Months Since Release of First OI

| OI A | B | C | D | E | J | O |
| 0 | 3 | 9 | 13 | 17 | 81 | 145 |
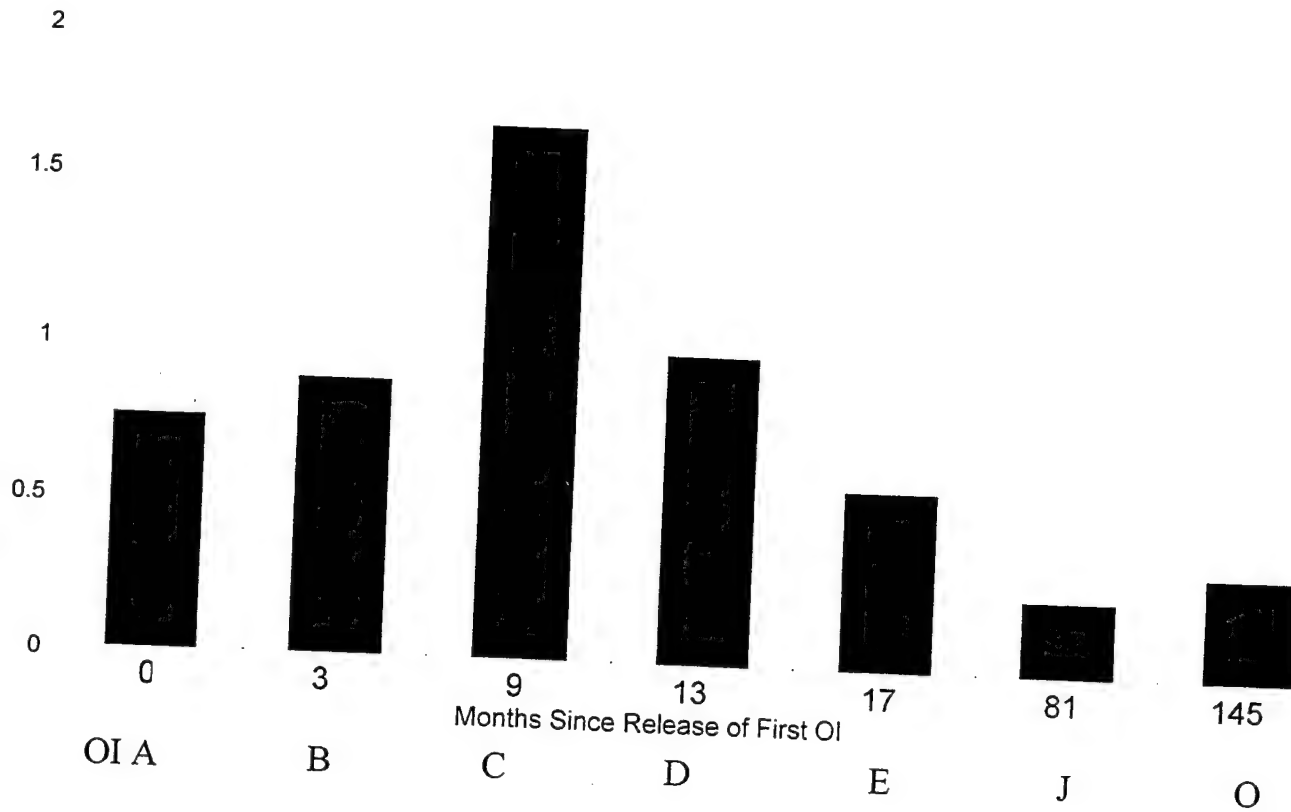
Figure 2. OID Failure Rate: Failures per Day

Decreasing Risk: More Test Time & Lower Risk Factors

Percent of Test Time
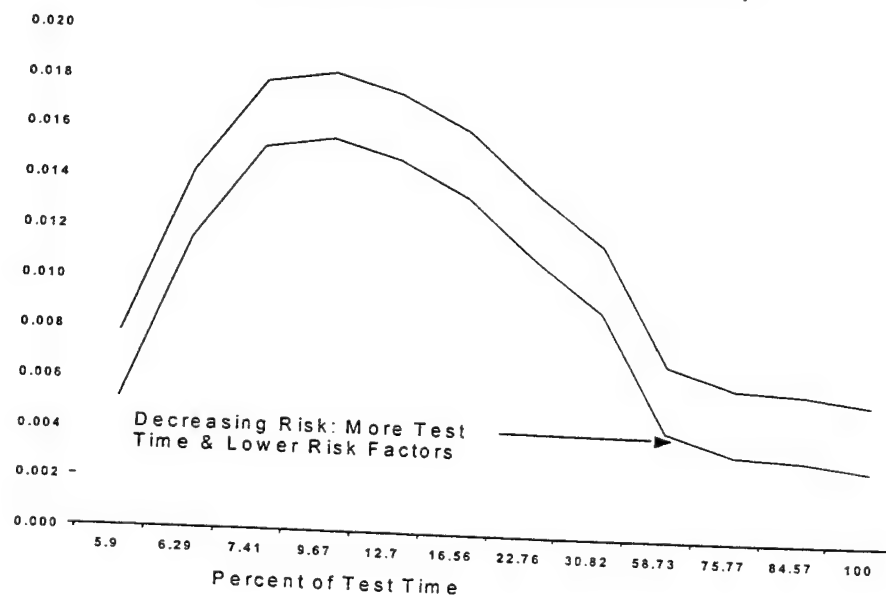
Reliability Modeling for Safety Critical Software

Norman F. Schneidewind, Fellow IEEE

Code SM/Ss
Naval Postgraduate School
Monterey, CA 93943, U.S.A.

Voice: (408) 656-2719
Fax : (408) 656-3407
Internet: schneidewind@nps.navy.mil

Keywords: software reliability prediction, safety critical software, risk analysis.

Summary and Conclusions

We show how software reliability predictions can increase confidence in the reliability of safety critical software such as the *NASA Space Shuttle Primary Avionics Software System* (*Shuttle* flight software). This objective was achieved using a novel approach to integrate software safety criteria, risk analysis, reliability prediction, and stopping rules for testing. This approach is applicable to other safety critical software. We only cover the safety of the software in a safety critical system. The hardware and human operator components of such systems are not explicitly modeled nor are the hardware and operator induced software failures. Our concern is with reducing the risk of all failures *attributed* to software. Thus, our use of the word *safety* refers to *software safety* and not to system safety. By improving the reliability of the software, where the reliability measurements and predictions are *directly related to mission and crew safety*, we contribute to system safety.

Remaining failures, *maximum failures*, *total test time* required to attain a given *fraction of remaining failures*, and *time to next failure* are shown to be useful reliability measurements and predictions for: 1) providing confidence that the software has achieved safety goals; 2) rationalizing how long to test a piece of software; and 3) analyzing the risk of not achieving *remaining failure* and *time to next failure* goals. Having predictions of the extent that the software is not fault free (*remaining failures*) and whether it is likely to survive a mission (*time to next failure*) provide criteria for assessing the risk of deploying the software. Furthermore, *fraction of remaining failures* can be used as both an *operational quality* goal in predicting *total test time* requirements and, conversely, as an indicator of *operational quality* as a function of *total test time* expended.

Software reliability models provide one of several tools that software managers of the *Shuttle* flight software are using to provide confidence that the software meets required safety goals. Other tools are inspections, software reviews, testing, change control boards, and perhaps most important -- experience and judgement.

1. Introduction

We propose that two categories of software reliability measurements (i.e., observed failure data used for model parameter estimation) and predictions (i.e., forecasts of future reliability using the parameterized model) be used in combination to *assist* in assuring the safety of the software in safety critical systems like the *Shuttle* flight software. The two categories are: 1) measurements and predictions that are associated with residual software faults and failures,

and 2) measurements and predictions that are associated with the ability of the software to survive a mission without experiencing a serious failure. In the first category are: *remaining failures, maximum failures, fraction of remaining failures,* and *total test time required to attain a given number or fraction of remaining failures.* In the second category are: *time to next failure* and *total test time required to attain a given time to next failure.* In addition, we define the risk associated with not attaining the required *remaining failures* and *time to next failure.* Lastly, we derive a quantity from the *fraction of remaining failures* that we call *operational quality.*

The benefits of predicting these quantities are: 1) they provide confidence that the software has achieved safety goals, and 2) they provide a means of rationalizing how long to test a piece of software (stopping rule). Having predictions of the extent that the software is not fault free (*remaining failures*) and its ability to survive a mission (*time to next failure*) are meaningful for assessing the risk of deploying safety critical software. In addition, with this type of information a software manager can determine whether more testing is warranted or whether the software is sufficiently tested to allow its release or unrestricted use. These predictions, in combination with other methods of assurance, such as inspections, defect prevention, project control boards, process assessment, and fault tracking, provide a quantitative basis for achieving safety and reliability goals [3].

Risk in the Webster's New Universal Unabridged Dictionary is defined as: "the chance of injury; damage, or loss" [19]. Some authors have extended the dictionary definition as

follows: "Risk Exposure=Probability of an Unsatisfactory Outcome*Loss if the Outcome is Unsatisfactory" [2]. Such a definition is frequently applied to the risks in managing software projects such as budget and schedule slippage. In contrast, our application of the dictionary definition pertains to the risk of executing the software of a safety critical system where there is the chance of injury (e.g., astronaut injury or fatality), damage (e.g., destruction of the *Shuttle*), or loss (e.g., loss of the mission) if a serious software failure occurs during a mission. We have developed risk criterion metrics to quantify the degree of risk associated with such an occurrence.

Lockheed-Martin, the primary contractor on the *Shuttle* flight software project, is experimenting with a promising algorithm which involves the use of the *Schneidewind Software Reliability Model* to compute a parameter: *fraction of remaining failures* as a function of the archived failure history during test and operation [10]. Our prediction methodology uses this parameter and other reliability quantities to provide bounds on *total test time, remaining failures, operational quality,* and *time to next failure* that are necessary to meet *Shuttle* safety requirements. We also show that there is a pronounced asymptotic characteristic to the *total test time* and *operational quality* curves that indicate the possibility of big gains in reliability as testing continues; eventually the gains become marginal as testing continues. We conclude that the prediction methodology is feasible for the *Shuttle* and other safety critical systems.

We only cover the safety of the software in a safety critical system. The hardware and human operator components of such systems are not explicitly modeled nor are the hardware and operator induced software failures. However, in practice, these hardware-software interface and human operator-software interface failures may be very difficult to identify as such; these failures may be recorded as software failures. Our concern is with reducing the risk of all failures *attributed* to software. Thus, our use of the word *safety* refers to *software safety* and not to system safety.

Although *remaining failures* has been discussed in general as a type of software reliability prediction [13], and various stopping rules for testing have been proposed, based on costs of testing and releasing software [4, 5, 8, 17], *failure intensity* [12], and testability [18], our approach is novel because we integrate software safety criteria, risk analysis, reliability prediction, and a stopping rule for testing. For a system like the *Shuttle,* where human lives are at risk, we cannot use economic or time-to-market criteria to determine when to deploy the software. Although *failure intensity* has proven useful for allocating test effort and determining when to stop testing in commercial systems [12], this criterion is not directly related to software safety. In a safety critical system, the prediction of remaining failures and identification of the faults which cause them is more relevant to ensuring safety than the trend of failure intensity over time. The latent faults must be found and removed through additional testing, inspection, or other means, if the safety of the mission is not to be jeopardized. Furthermore, as we will show, *remaining failures*, along with *time to next failure*, can be used

as risk criteria. It is not clear how *failure intensity* could be a meaningful safety criterion.

Because *testability* attempts to quantify the probability of failure, if the code is faulty [18], this criterion has a relationship with reliability if we *know* that the code is faulty. However in the *Shuttle* and other safety critical software, our purpose is to *predict* whether the code is faulty. For safety critical software, we must use reliability measurements and predictions to assess whether safety and mission goals are likely to be achieved.

We first define two criteria for software safety. Then we apply these criteria to risk analysis of safety critical software, using the *Shuttle* flight software as an example. Next, we define and provide brief derivations for a variety of prediction equations that are used in reliability prediction and risk analysis; included is the relationship between *time to next failure* and *reduction in remaining failures*. This is followed by an explanation of the principal of *optimal selection of failure data* that involves selecting only the most relevant set of failure data for reliability prediction, with the result of producing more accurate predictions than would be the case if the entire set of data were used. Then we show how the prediction equations can be used to integrate testing with reliability and quality. An example is shown of how the risk analysis and reliability predictions can be used to make decisions about whether the software is safe to deploy. Lastly we show validation results for a variety of predictions.

Acronyms

OIA: *Shuttle* operational increment A

OIB: *Shuttle* operational increment B

OIC: *Shuttle* operational increment C

OID: *Shuttle* operational increment D

Assumptions [1]:

1. Faults that cause failures are removed.

2. As more failures occur and more faults are corrected, *remaining failures* will be reduced.

3. The *remaining failures* are "zero" for those OI's that were executed for extremely long times (years) with no additional failure reports; correspondingly, for these OI's, maximum failures equals total observed failures.

4. The number of failures detected in one interval is independent of the failure count in another.

5. Only "new" failures are counted (i.e., failures that are repeated as a consequence of not correcting a fault are not counted).

orrecting a fault are not counted).

Definitions

o *Interval*: an integer time unit t of constant length defined by $t-1<t<t+1$, where $t>0$; failures are counted in intervals (e.g., one failure occurred in *interval* 4) [1, 7].

o *Number of Intervals*: the number of contiguous integer time units t of constant length represented by a positive real number (e.g., the predicted time to next failure is *3.87 intervals*).

o *Operational Increment* (OI): a software system comprised of modules and configured from a series of builds to meet *Shuttle* mission functional requirements.

o *Time*: Continuous CPU execution time over an *interval* range.

Severity Codes:

1. Severe Vehicle or Crew Performance Implications.

2. Affects Ability to Complete Mission (Not a safety issue).

3. Workaround Available, Minimal Effect on Procedures.

4. Insignificant (Paperwork, etc.).

5. Not Visible to User.

Nomenclature

o *Predicted at time t*: a prediction made in the *interval* t.

o *Safety*: software safety; not system safety.

Notation

α	failure rate at the beginning of interval s
β	negative of derivative of failure rate divided by failure rate (i.e., relative failure rate)
F(i)	predicted failure count in the range [1,i]; used in computing MSE_r
F_{ij}	observed failure count during interval j since interval i; used in computing MSE_T
F(t)	predicted failure count in the range [1, t]

F_t given number of failures to occur after interval t; used in predicting $T_F(t)$

$F(t_1, t_2)$ predicted failure count in the range $[t_1, t_2]$

$F(\infty)$ predicted failure count in the range $[1, \infty]$; maximum failures over the life of the software

i current interval

j next interval j>i where $F_{ij}>0$

J maximum j≤t where $F_{ij}>0$.

MSE_F mean square error criterion for selecting s for failure count predictions

MSE_r mean square error criterion for selecting s for remaining failure predictions

MSE_T mean square error criterion for selecting s for time to next failure predictions

p(t) fraction of remaining failures predicted at time t

Q(t) operational quality predicted at time t; the complement of p(t); the degree to which software is free of remaining faults (failures)

r_c critical value of remaining failures; used in computing RCM $r(t_t)$

r(t) remaining failures predicted at time t

$r(t_t)$ remaining failures predicted at total test time t_t

$\Delta r(T_F, t)$ reduction in remaining failures that would be achieved if the software were executed for a time T_F, predicted at time t

RCM $r(t_t)$ risk criterion metric for remaining failures at total test time t_t

RCM $T_F(t_t)$ risk criterion metric for time to next failure at total test time t_t

| s | starting interval for using observed failure data in parameter estimation |

s^* — optimal starting interval for using observed failure data, as determined by MSE criterion

t — cumulative time in the range [1,t]; last interval of observed failure data; current interval

t_m — mission duration (end time-start time); used in computing RCM $T_F(t_t)$

t_t — total test time (observed or predicted)

$T_F(t)$ — time to next failure(s) predicted at time t

$T_F(t_t)$ — time to next failure predicted at total test time t_t

$T_F(\Delta r, t)$ — time to next N failures that would be achieved if remaining failures were reduced by Δr, predicted at time t

T_{ij} — time since interval i to observe number of failures F_{ij} during interval j; used in computing MSE_T^i

X_i — observed failure count in the range [1,i]

X_{s-1} — observed failure count in the range [1,s-1]

$X_{s,t}$ — observed failure count in the range [s,t]

$X_{s,t1}$ — observed failure count in the range [s,t_1]

X_t — observed failure count in the range [1,t]

X_{t1} — observed failure count in the range [1,t_1]

2. Criteria for Safety

If we define our safety goal as the reduction of failures that would cause loss of life, loss of mission, or abort of mission to an acceptable level of risk [11], then for software to be ready to deploy, after having been tested for total time t_t, we must satisfy the following criteria:

1) predicted remaining failures $r(t_t) < r_c$, $\qquad\qquad\qquad\qquad\qquad\qquad$ (1)

where r_c is a specified critical value , and

2) predicted time to next failure $T_F(t_t) > t_m$, $\qquad\qquad\qquad\qquad\qquad$ (2)

where t_m is mission duration.

For systems that are tested and operated continuously like the *Shuttle*, t_t, $T_F(t_t)$, and t_m are measured in execution time. Note that, as with any methodology for assuring software safety, we cannot guarantee safety. Rather, with these criteria, we seek to reduce the risk of deploying the software to an acceptable level.

2.1 Remaining Failures Criterion

Using *assumption 1* that the faults that cause failures are removed (this is the case for the *Shuttle*), *criterion 1* specifies that the residual failures and faults must be reduced to a level where the risk of operating the software is acceptable. As a practical matter, we suggest $r_c = 1$. That is, the goal would be to reduce the expected *remaining failures* to less than one before deploying the software. The reason for this choice is that one or more *remaining failures* would constitute unacceptable risk for safety critical systems. This is the threshold used by the *Shuttle* software managers. One way to specify r_c is by failure severity level (e.g., *severity*

224

level 1 for life threatening failures). Another way, which imposes a more demanding safety requirement, is to specify that r_c represents *all* severity levels. For example, $r(t_t)<1$ would mean that $r(t_t)$ must be less than one failure, *independent* of severity level.

If we predict $r(t_t) \geq r_c$, we would continue to test for a total time $t_t' > t_t$ that is predicted to achieve $r(t_t')<r_c$, using *assumption 2* that we will experience more failures and correct more faults so that the *remaining failures* will be reduced by the quantity $r(t_t)-r(t_t')$. If the developer does not have the resources to satisfy the criterion or is unable to satisfy the criterion through additional testing, the risk of deploying the software prematurely should be assessed (see the next section). We know from Dijkstra's dictum that we cannot demonstrate the absence of faults [6]; however we can reduce the risk of failures occurring to an acceptable level, as represented by r_c. This scenario is shown in Figure 1. In *case A* we predict $r(t_t)<r_c$ and the mission begins at t_t. In *case B* we predict $r(t_t) \geq r_c$ and postpone the mission until we test for total time t_t' and predict $r(t_t')<r$. In both cases *criterion 2)* must also be satisfied for the mission to begin.

2.2 Time to Next Failure Criterion

Criterion 2 specifies that the software must survive for a time greater than the duration of the mission. If we predict $T_F(t_t) \leq t_m$, we would continue to test for a total time $t_t''>t_t$ that is predicted to achieve $T_F(t_t'')>t_m$, using *assumption 2* that we will experience more failures and correct more faults so that the *time to next failure* will be increased by the quantity $T_F(t_t'')-T_F(t_t)$. Again, if it is infeasible for the developer to satisfy the criterion for lack of resources or

failure to achieve test objectives, the risk of deploying the software prematurely should be assessed (see the next section). This scenario is shown in Figure 2. In *case A* we predict $T_F(t_t) > t_m$ and the mission begins at t_t. In *case B* we predict $T_F(t_t) \leq t_m$ and postpone the mission until we test for total time t_t'' and predict $T_F(t_t'') > t_m$. In both cases *criterion 1)* must also be satisfied for the mission to begin. If neither criterion is satisfied, we test for a time which is the greater of t_t' or t_t''.

3. Risk Assessment

The amount of *total test time* t_t can be considered a measure of the degree to which software reliability goals have been achieved. This is particularly the case for systems like the *Shuttle* where the software is subjected to continuous and rigorous testing for several years in multiple facilities, using a variety of operational and training scenarios (e.g., by Lockheed-Martin in Houston, by NASA in Houston for astronaut training, and by NASA at Cape Kennedy). If we view t_t as an input to a risk reduction process, and $r(t_t)$ and $T_F(t_t)$ as the outputs, we can portray the process as shown in Figure 3, where r_c and t_m are shown as "risk criteria levels" of safety that control the process. While we recognize that *total test time* is not the only consideration in developing test strategies and that there are other important factors, like the consequences for reliability and cost, in selecting test cases [20], nevertheless, for the foregoing reasons, *total test time* has been found to be strongly positively correlated with reliability growth for the *Shuttle* [15].

3.1 Remaining Failures

We can formulate the mean value of the *risk criterion metric* (RCM) for *criterion 1* as follows:

$$\text{RCM } r(t_t) = (r(t_t) - r_c)/r_c = (r(t_t)/r_c) - 1 \tag{3}$$

We plot equation (3) in Figure 4 as a function of t_t for $r_c = 1$, where *positive, zero,* and *negative* values correspond to $r(t_t) > r_c$, $r(t_t) = r_c$, and $r(t_t) < r_c$, respectively. In Figure 4, these values correspond to the following regions: *UNSAFE* (i.e., above the X-axis predicted *remaining failures* are greater than the "safe" value); *NEUTRAL* (i.e., on the X-axis predicted *remaining failures* equal to the "safe" value); and *SAFE* (i.e., below the X-axis predicted *remaining failures* are less than the "safe" value).

This graph is for the *Shuttle operational increment OID*. In this example we see that at approximately $t_t = 57$ the risk transitions from the *UNSAFE* region to the *SAFE* region.

3.2 Time to Next Failure

Similarly, we can formulate the mean value of the *risk criterion metric* (RCM) for *criterion 2* as follows:

$$\text{RCM } T_F(t_t) = (t_m - T_F(t_t))/t_m = 1 - (T_F(t_t))/t_m \tag{4}$$

We plot equation (4) in Figure 5 as a function of t_t for $t_m = 8$ days (a typical mission duration time for this OI), where *positive, zero,* and *negative* risk corresponds to $T_F(t_t) < t_m$, $T_F(t_t) = t_m$, and $T_F(t_t) > t_m$, respectively. In Figure 5, these values correspond to the following regions: *UNSAFE* (i.e., above the X-axis predicted *time to next failure* is less than the "safe" value); *NEUTRAL* (i.e., on the X-axis predicted *time to next failure* is equal to the "safe" value); and

SAFE (i.e., below the X-axis predicted *time to next failure* is greater than the "safe" value). This graph is for the *Shuttle operational increment OIC.* In this example we see that at all values of t_t the RCM is in the *SAFE* region.

4. Approach to Prediction

In order to support our safety goal and to assess the risk of deploying the software, we make various reliability and quality predictions. In addition, we use these predictions to perform tradeoff analysis between reliability and *total test time*. Thus, our approach is to use a software reliability model to predict the following: 1) *maximum failures, remaining failures,* and *operational* quality (as defined in the next section); 2) *time to next failure* (beyond the last observed failure); 3) *total test time* necessary to achieve required levels of *remaining failures* (fault) level, *operational quality,* and *time to next failure;* and 4) tradeoffs between increases in levels of reliability and quality with increases in testing.

5. Prediction Equations

The following prediction equations are based on the *Schneidewind Software Reliability Model* [1, 14, 15, 16], one of the four models recommended in the *AIAA Recommended Practice for Software Reliability* [1].These equations use *assumptions 4-7* in the *Introduction.*

We derive these equations in the next section. . We apply them to analyze the reliability of the *Shuttle* flight software. All predictions are mean values.

Because the flight software is run continuously, around the clock, in simulation, test, or flight, "time" refers to continuous execution time and *total test time* refers to execution time that is used for testing. Failure count intervals are equal to 30 days of continuous execution time. This interval is long because the *Shuttle* software is tested for several years; a 30 day interval length is a convenient for recording failures for software that is tested this long.

In the following equations, the parameter α is the failure rate at the beginning of interval s; the parameter β is the negative of derivative of failure rate divided by failure rate (i.e., relative failure rate); t is the last interval of observed failure data; s is the starting interval for using observed failure data in parameter estimation that will result in the best estimates of α and β and the most accurate predictions [14]; X_{s-1} is the observed failure count in the range [1,s-1]; $X_{s,t}$ is the observed failure count in the range [s,t]; and $X_t = X_{s-1} + X_{s,t}$. These failure count interval relationships are shown in Figure 6; also shown is *total test time* t_t. Failures are counted against *operational increments* (OIs). Data from four *Shuttle* OI's, designated *OIA*, *OIB, OIC*, and *OID* are used in this analysis.

5.1 Cumulative Failures

When maximum likelihood estimates are obtained for the parameters α and β, with s as the starting interval for using observed failure data, we obtain the predicted *failure count in the range [s,t]*:

$$F_{s,t}=(\alpha/\beta)[1-\exp(-\beta((t-s+1)))] \tag{5}$$

Furthermore, if we add X_{s-1}, the observed failure count in the range [1,s-1], we obtain predicted *failure count in the range [1, t]*:

$$F(t)=(\alpha/\beta)[1-\exp(-\beta((t-s+1)))]+X_{s-1} \tag{6}$$

5.2 Failures in an Interval Range

If we set $t\equiv t_2$ and subtract $X_{t1}=X_{s-1}+X_{s,t1}$, the observed failure count in the range [1,t_1], from equation (6), we obtain the predicted *failure count in the range* $[t_1,t_2]$:

$$F(t_1,t_2)=(\alpha/\beta)[1-\exp(-\beta((t_2-s+1)))]-X_{s,t1} \tag{7}$$

5.3 Maximum Failures

If we let $t\to\infty$ in equation (6), we obtain the predicted *failure count in the range $[1,\infty]$* (i.e., *maximum failures* over the life of the software):

$$F(\infty)=\alpha/\beta+X_{s-1} \tag{8}$$

5.4 Remaining Failures

To obtain predicted *remaining failures* $r(t)$ at time t, we subtract $X_t=X_{s-1}+X_{s,t}$ from equation (8):

$$r(t)=(\alpha/\beta)-X_{s,t}=F(\infty)-X_t \tag{9}$$

$r(t)$ can also be expressed as a function of *total test time* t_t by substituting equation (5) into equation (9) and setting $t\equiv t_t$:

$$r(t_t)=(\alpha/\beta)(\exp-\beta[t_t-(s-1)]) \tag{10}$$

5.5 Fraction of Remaining Failures:

If we divide equation (9) by equation (8), we obtain *fraction of remaining failures* predicted at time t:

$$p(t)=r(t)/F(\infty) \tag{11}$$

5.6 Operational Quality

The *operational quality* of software is the complement of $p(t)$. It is the degree to which software is free of remaining faults (failures), using *assumption 1* that the faults that cause failures are removed. It is predicted at time t as follows:

$$Q(t)=1-p(t) \tag{12}$$

5.7 Total Test Time to Achieve Specified Remaining Failures

The predicted *total test time* required to achieve a specified *number of remaining failures*

$$t_t = [\log[\alpha/(\beta[r(t_t)])]]/\beta + (s1)$$

at t_t, $r(t_t)$, is obtained from equation (10) by solving for t_t:

5.8 Time to Next Failure

By substituting $t_2=t+T_F(t)$ in equation (7), setting $t_1 \equiv t$, defining $F_t=F(t,t+T_F)$, and solving for $T_F(t)$, we obtain the predicted *time for the next F_t failures* to occur, when the current time

$$T_F(t) = [(\log[\alpha/(\alpha\beta(X_{s,t}+F_t))])/\beta](ts+1)$$

$$\text{for } (\alpha/\beta) > (X_{s,t}+F_t)$$

is t :

The terms in $T_F(t)$ have the following definitions:

231

t: Current interval;

$X_{s,t}$:   Observed failure count in the range [s,t]; and

F_t: Given number of failures to occur after interval t.

We consider equations (5)-(11) and (14) to be predictors of reliability that are related to safety; equation (13) represents the predicted *total test time* required to achieve stated safety goals. If a quality requirement is stated in terms of *fraction of remaining failures*, the definition of Q as *Operational Quality*, equation (12), is consistent with the IEEE definition of quality: the *degree* to which a system, component, or process meets specified requirements [9]. For example, if a reliability specification requires that software is to have no more that 5% remaining failures (i.e., p=.05, Q=.95) after testing for a total of t_t intervals, then a predicted Q of .90 would indicate the *degree* to which the software meets specified requirements.

5.9 Relating Time to Next N Failures and Remaining Failures Predictions

Although we have shown the risk analysis and prediction equations for *remaining failures* and *time to next failure* separately, it would be useful to combine these quantities in one equation so that we can predict the effect on one quantity for a given change in the other. In particular we want to predict, at time t, the *time to the next N failures*, $T_F(\Delta r, t)$, that would be achieved if *remaining failures* were reduced by Δr. We use *assumption 1* that $N=\Delta r$; that is, faults that cause failures are removed. When N=1, we have the familiar *time to next failure*. When N>1, $T_F(\Delta r, t)$ is interpreted as cumulative execution time for the N failures to occur.

Conversely, we want to predict, at time t, the *reduction in remaining failures*, $\Delta r(T_F,t)$, that would be achieved if the software were executed for a time T_F. This relationship is derived by using equation (10) and setting $\Delta r=r(t_1)-r(t_t)$, $t_t=t_1+\Delta t$, and $t_1 \equiv t$, and solving for $\Delta t \equiv T_F(\Delta r,t)$:

$$T_F(\Delta r,t)=(-1/\beta)[\log[1-((\beta \Delta r/\alpha)(\exp(\beta(t-s+1))))]] \tag{15}$$

for $((\beta \Delta r/\alpha)(\exp(\beta(t-s+1))))<1$.

Equation (15) is analogous to equation (14). Also, Δr in equation (15) is analogous to F_t in equation (14), if we use *assumption 1* that the faults that cause the F_t failures are removed, with a corresponding *reduction in remaining failures*. The two equations produce the same result for the same parameter values. Equation (15) has the advantage of being a simpler computation because it does not require the observed data vector $X_{s,t}$, which is used in equation (14). Also, equation (15) is convenient to use for trading off *time to next N failures* against *reduction in remaining failures*, and the effort and the *total test time* implicit in making the reductions.

We can invert equation (15) to solve for the *reduction in remaining failures* that would be achieved by executing the software for a time T_F.

$$\Delta r(T_F,t)=(\alpha/\beta)[\exp(-\beta(t-s+1))][1-\exp(-\beta(T_F))] \tag{16}$$

6. Criterion for Optimally Selecting Failure Data

The first step in identifying the optimal value of s (s^*) is to estimate the parameters α and β for each value of s in the range [1,t] where convergence can be obtained [1, 14, 16]. Then the *Mean Square Error* (MSE) criterion is used to select s^*, the failure count interval that

corresponds to the minimum MSE between predicted and actual failure counts (MSE_F), *time to next failure* (MSE_T), or *remaining failures* (MSE_r), depending on the type of prediction. The first two were reported in [14]. In this paper we develop MSE_r. MSE_r is also the criterion for *maximum failures* ($F(\infty)$) and *total test time* (t_t) because the two are functionally related to *remaining failures* ($r(t)$); see equations 9 and 13. We also show MSE_T because it is used in predictions that involve *time to next failure*: $T_F(t)$, $T_F(\Delta r, t)$, and $\Delta r(T_F, t)$. Once α, β, and s are estimated from observed counts of failures, the foregoing predictions can be made. The reason MSE is used to evaluate which triple (α, β, s) is best in the range [1,t] is that research has shown that because the product and process change over the life of the software, old failure data (i.e., s=1) are not as representative of the current state of the product and process as the more recent failure data (i.e., s>1) [14]. The optimal values of s (s^*) that were used in the risk analysis and prediction examples are shown in Tables 1-4.

The *Statistical Modeling and Estimation of Reliability Functions for Software* (SMERFS) [7] is used for all predictions except t_t, $T_F(\Delta r, t)$, and $\Delta r(T_F, t)$, which are not implemented in SMERFS.

6.1 Mean Square Error Criterion for Remaining Failures

Although we can never know whether additional failures may occur, nevertheless we can form the difference between two equations for $r(t)$: (9), which is a function of predicted *maximum failures* and the observed failures, and (10), which is a function of *total test time*, and apply the MSE criterion. This yields the following Mean Square Error (MSE_r) criterion

$$MSE_r = \frac{\sum\limits_{i=s}^{t}[F(i)X_i]^2}{ts+1}$$

for number of *remaining failures*:

where $F(i)$ is the predicted *failure count in the range [1,i]* and X_i is the observed failure count in the range $[1,i]$.

6.2 Mean Square Error Criterion for Time to Next Failure(s)

The Mean Square Error (MSE_T) criterion for *time to next failure*(s), which was derived in

$$MSE_T = \frac{\sum\limits_{i=s}^{J-1}[[\log[\alpha/(\alpha-\beta(X_{s,i}+F_{ij}))]/\beta(is+1)]-T_{ij}]^2}{(J-s)}$$

$$\text{for } (\alpha/\beta) > (X_{s,i}+F_{ij})$$

[14], is given by equation (18):

The terms in MSE_T have the following definitions:

i: Current interval;

j: Next interval j>i where $F_{ij}>0$;

$X_{s,i}$: Observed failure count in the range $[s,i]$;

F_{ij}: Observed failure count during interval j since interval i;

T_{ij}: Time since i to observe number of failures F_{ij} during j (i.e., T_{ij}=j-i)

t: The last interval of observed failure data; and

J: Maximum $j \leq t$ where $F_{ij} > 0$.

7. Relating Testing to Reliability and Quality

7.1 Predicting Total Test Time and Remaining Failures

We use equation (8) to predict *maximum failures* ($F(\infty) = 11.76$) for *Shuttle OIA*. Using given values of p and equation (11) and setting $t \equiv t_t$, we predict $r(t_t)$ for each value of p. The values of $r(t_t)$ are the predictions of *remaining failures* after the OI has been executed for *total test time* t_t. Then we use the values of $r(t_t)$ and equation (13) to predict corresponding values of t_t. The results are shown in Figure 7, where $r(t_t)$ and t_t are plotted against p for *OIA*. Note that required *total test time* t_t rises very rapidly at small values of p and $r(t_t)$. Also note that the maximum value of p on the plot corresponds to $t_t = 18$ and that smaller values correspond to *future* values of t_t (i.e., $t_t > 18$).

7.2 Predicting Operational Quality

Equation (12) is a useful measure of the *operational quality* of software because it measures the degree to which faults have been removed from the software (using *assumption 1* that the faults that cause failures are removed), relative to predicted *maximum failures*. We call this type of quality *operational* (i.e., based on executing the software) to distinguish it from static quality (e.g., based on the complexity of the software).

Using given values of p and equations (11) and (12)and setting $t \equiv t_t$, we compute $r(t_t)$ and Q, respectively. The values of $r(t_t)$ are then used in equation (13) to compute t_t. The corresponding values of Q and t_t are plotted in Figure 8 as *Operational Quality* and *Total Test*

Time, respectively for *OIA*. We again observe the asymptotic nature of the testing relationship in the great amount of testing required to achieve high levels of quality.

7.3 Predicting Time to Next Failure

First, we show the actual *time to next failure* in Figure 9 for *OIA* on the solid curve that has occurred in the *execution time* range t=[1,18], where one failure occurred at t=4, 14, and 18, and two failures occurred at t=8 and 10. All failures were *Severity Level* 3: "Workaround available; minimal effect on procedures". The way to read the graph is as follows: If we take a given failure, *Failure 1*, for example, it occurs at t=4; therefore, at t=1 the *time to next failure*=3 (4-1); at t=2 the *time to next failure*=2 (4-2); at t=4 *Failure 1* occurs, so the *time to next failure*=4 (8-4) now refers to *Failure 2*, etc. Next, using equation (14), we predict the *time to next failure* $T_F(18)$ to be 4 (3.87 rounded) on the dashed curve. Based on the foregoing, this prediction indicates we should continue testing if $T_F(18)=3.87 \le t_m$ (mission duration).

7.4 Predicting Tradeoffs of Time to Next N Failures with Reduced Remaining Failures

By using equation (15), we can predict *time to next N failures*, $T_F(\Delta r,t)$, as a function of *reduction in remaining failures*, Δr. This is shown in Figure 10 for *OIA* , where, for example, with $\Delta r=1$, we predict $T_F(1,18)=3.87$ (i.e., a *reduction in remaining failures* of 1 corresponds to achieving a *time to next failure* of 3.87 intervals from the current interval 18). Conversely, by using equation (16), we predict *reduction in remaining failures*, $\Delta r(T_F,t)$, as a function of *time to next failure*, T_F. This is shown in Figure 11 for *OIA*, where, for example, with

T_F=3.87, we predict $\Delta r(3.87,18)$=1 (i.e., executing *OIA* for a *time to next failure* of 3.87 intervals from the current interval 18 corresponds to achieving a *reduction in remaining failures* of 1). We provide further elaboration of these graphs in the next section.

8. Making Safety Decisions

In making the decision about how long to test, t_t, we apply our safety criteria and risk assessment approach. We use Table 1 to illustrate the process. For t_t=18 (when the last failure occurred on *OIA*), r_c=1, and t_m=8 days (.267 intervals), we show *remaining failures, RCM for remaining failures, time to next failure, RCM for time to next failure,* and *operational quality.* These results indicate that safety *criterion 2* is satisfied but not *criterion 1* (i.e., *UNSAFE* with respect to *remaining failures*); also *operational quality* is low.

By looking at Figure 10 and Table 1, we see that if we reduce *remaining failures* r(18) by 1 from 4.76 to 3.76 (non-integer values are possible because the predictions are mean values), the predicted *time to next failure* that would be achieved is T_F(18)=3.87 intervals. These predictions satisfy *criterion 2* (i.e., T_F(18)=3.87>t_m=.267) but not *criterion 1* (i.e., r(18)=4.76>r_c=1). Note also in Figure 10 and Table 1 that *fraction of remaining failures* p=1-Q=.40 at r(18)=4.76. Now, if we continue testing for a total time t_t=52 intervals, as shown in Figure 10 and Table 1, and reduce *remaining failures* from 4.76 to .60, the predicted *time to next 4.16 failures* that would be achieved is 33.94 (34, rounded) intervals. This corresponds to t_t=18+34=52 intervals. That is, if we test for an additional 34 intervals, starting at interval 18, we would expect to experience 4.16 failures. These predictions now satisfy *criterion 1*

because $r(52)=.60<r_c=1$. Note also in Figure 10 and Table 1 that *fraction of remaining failures* $p=1-Q=.05$ at $r(52)=.60$. Using the converse of the relationship in Figure 10, provides another perspective, as shown in Figure 11, where we see that if we continue to test for an additional $T_F=34$ intervals, starting at interval 18, the predicted *reduction in remaining failures* that would be achieved is 4.16 or $r(52)=.60$.

Lastly, Figure 12 shows the *Launch Decision,* relevant to the *Shuttle,* (or, generically, *the Deployment Decision*), where *remaining failures* are plotted against *total test time* for *OIA*. With these results in hand, the software manager can decide whether to deploy the software depending on factors such as predicted *remaining failures*, as shown in Figure 12, along with considering other factors such as the trend in reported faults over time, inspection results, etc.. If testing were to continue until $t_t=52$, the predictions in Figure 12 and Table 1 would be obtained. These results show that *criterion 1* is now satisfied *(i.e., SAFE)* and *operational quality* is high. We also see from Figure 12 that at this value of t_t, further increases in t_t would not result in a significant increase in reliability and safety. Also note that at $t_t=52$ it is not feasible to make a prediction of $T_F(52)$ because the predicted *remaining failures* is less than one.

Table 1

Safety Criteria Assessment

OIA

	$r_c=1$						$t_m=8$ days		
t_t	α	β	s^*	$r(t_t)$	RCM r(t)	s^*	$T_F(t_t)$	RCM $T_F(t_t)$	Q
18	.534	.061	9	4.76	3.76	9	3.87	-13.49	.60
52	.534	.061	9	.60	-.40	9	*	*	.95

30 day Total Test Time and Time to Next Failure Intervals.

* Cannot predict because predicted Remaining Failures is less than one.

9. Summary of Predictions and Validation

9.1 Predictions

Table 2 shows a summary of remaining and maximum failure predictions compared with actual failure data, where available, for *OIA, OIB, OIC*, and *OID*. Because we do not know the *actual* remaining and maximum failures, we use *assumption 3*: *remaining failures* are "zero" for those OI's (*B, C*, and *D*) that were executed for extremely long times (years) with

no additional failure reports; correspondingly, for these OI's, we use *assumption 3* that *maximum failures* equals total observed failures.

Table 2

Predicted Remaining and Maximum Failures versus Actuals

	t_t	s^*	α	β	$r(t_t)$	Actual r	$F(\infty)$	Actual F
OIA	18	9	.534	.061	4.76	?[A]	11.76	7[A]
OIB	20	1	1.69	.131	0.95	1[B]	12.95	13[B]
OIC	20	7	1.37	.126	1.87	2[C]	12.87	13[C]
OID	18	6	.738	.051	7.36	4[D]	17.36	14[D]

30 day Total Test Time Intervals

Time of last recorded failure:

A. No additional failures have been reported after 17.17 intervals.

B. The last recorded failure occurred at 63.67 intervals.

C. The last recorded failure occurred at 43.80 intervals.

D. The last recorded failure occurred at 65.03 intervals.

Table 3 shows a summary of *total test time* and *time to next failure* predictions compared

with actual execution time data, where available, for *OIA, OIB, OIC*, and *OID*.

Table 3

Predicted Total Test Time and Time to Next Failure versus Actuals

	s^*	$t_t(r=1)$	Actual t_t	t	s^*	$T_F(t)$	Actual T_F
OIA	9	43.59	?	18	9	3.9	?
OIB	1	*	63.67	20	*	*	43.67
OIC	7	24.98	27.07	20	5	4.2	7.63
OID	6	56.84	58.27	18	5	6.4	6.2

30 day Total Test Time and Time to Next Failure Intervals.

* Cannot predict because predicted Remaining Failures is less than one.

Additional Predictions for OID:

The following are additional predictions of total test time for OID that are not listed in Table 3: $t_t(r=2)=43.35$, Actual=45.17; $t_t(r=3)=35.47$, Actual=23.70.

Table 4 shows a summary of the predictions of *time to next failure* for a given *reduction in remaining failures* of 1 and the predictions of *reduction in remaining failures* for given *time to next failure* compared with actual execution time and failure data, where available, for *OIA, OIB, OIC*, and *OID*.

Table 4

Predicted Tradeoffs of Time to Next Failure with Reduced Remaining Failures

versus Actuals

	t	s*	α	β	$T_F(\triangle r=\underline{1},t)$	Actual	(T_F,t)	$\triangle r(T_F,t)$	Actual
OIA	18	9	.534	.061	3.87	?	3.87	1.00	?
OIB	20	1	1.69	.131	*	43.67	43.67	.95	1.0
OIC	20	5	1.34	.096	4.16	7.63	7.63	1.58	1.0
OID	18	5	1.61	.137	6.35	6.20	6.20	.99	1.0

30 day Total Test Time and Time to Next Failure Intervals.

* Cannot predict because predicted Remaining Failures is less than one.

9.2 Validation

A total of 18 predictions were made across Tables 2, 3, and 4, where there was an actual value to compare: three r(t), four F(∞), four t_t, two T_F(t), two T_F(△r,t), and three △r(T_F,t). The mean relative error (mean of (actual-predicted)/actual) of prediction is 22.92% and the standard deviation is 27.61%. In making these predictions we note both the sparsity of post-delivery failures and the extremely long test times for *Shuttle* flight software, as summarized in Table 5. See the *Appendix* for a listing of the failure data. Despite the fact that the *Schneidewind Software Reliability Model* uses optimal selection of failure data, and thus less than the full set of data, there must be a minimum number of failures to *start the parameter*

estimation process, understanding that the model will then select the optimal value of s(s*).

Thus, given the sparsity of the data, all failures in Table 5 were used in parameter estimation, regardless of their severity. Furthermore, as described earlier, a more conservative risk assessment is produced if all categories of failures are included in the analysis.

Table 5

Failure Distribution by Severity Code

	Severity 2 Failures	Severity 3 Failures	Severity 4 Failures	Maximum Failures	Total Test Time
OIA	0	7	0	7	18
OIB	5	8	0	13	64
OIC	3	6	2	13*	44
OID	6	8	0	14	66

30 day Total Test Time Intervals.

* Unknown Severity for two failures

There are no post-delivery Severity 1 or 5 failures in the above Operational Increments.

Observed Failure Counts

(Interval i = 30 days execution time)

i	OIA	OIB	OIC	OID
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	1	2	0	0
5	0	1	0	3
6	0	0	2	1
7	0	0	1	0
8	2	2	3	1
9	0	1	1	0
10	2	0	0	1
11	0	2	0	1
12	0	0	0	0
13	0	1	1	2
14	1	0	1	0
15	0	0	0	0
16	0	0	0	0
17	0	0	1	0
18	1	0	0	1
19		0	0	0
20		0	1	0
21		0	0	0
22		0	0	0
23		0	0	0
24		0	0	1
25		0	0	0
26		0	0	0
27		0	0	0
28		0	1	0
29		0	0	0
30		0	0	0
31-63		0		
64		1		
31-43			0	
44			1	
31-45				0
46				1
47-58				0
59				1
60-65				0
66				1
Totals:	7	13	13	14

Acknowledgments

References

[1] Recommended Practice for Software Reliability, R-013-1992, *American National Standards Institute/American Institute of Aeronautics and Astronautics*, 370 L'Enfant Promenade, SW, Washington, DC 20024, 1993.

[2] Barry W. Boehm, "Software Risk Management: Principles and Practices", *IEEE Software*, Vol. 8, No. 1, January 1991, pp. 32-41.

[3] C. Billings, et al, "Journey to a Mature Software Process", *IBM Systems Journal*, Vol. 33, No. 1, 1994, pp. 46-61.

[4] Siddhartha R. Dalal and Allen A. McIntosh, "When to Stop Testing for Large Software Systems with Changing Code", *IEEE Transactions on Software Engineering*, Vol. 20, No. 4, April 1994, pp. 318-323.

[5] Siddhartha R. Dalal and Allen A. McIntosh, "Some Graphical Aids for Deciding When to Stop Testing", *IEEE Journal on Selected Areas in Communications*, Vol. 8, No.2, February 1990, pp. 169-175.

[6] E. W. Dijkstra, "Structured Programming", *Software Engineering Techniques*, eds. J. N. Buxton and B. Randell, NATO Scientific Affairs Division, Brussels 39, Belgium, April 1970 pp. 84-88.

[7] William H. Farr and Oliver D. Smith, *Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide*. NAVSWC TR-84-373, Revision 3, Naval Surface Weapons Center, Revised September 1993.

[8] Willa Ehrlich, et al, "Determining the Cost of a Stop-Test Decision", *IEEE Software*, March 1993, pp. 33-42.

[9] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12.1990, The Institute of Electrical and Electronics Engineers, New York, New York, March 30, 1990.

[10] Ted Keller, Norman F. Schneidewind, and Patti A. Thornton "Predictions for Increasing Confidence in the Reliability of the Space Shuttle Flight Software", *Proceedings of the AIAA Computing in Aerospace 10*. San Antonio, TX, March 28, 1995, pp. 1-8.

[11] Nancy G. Leveson, "Software Safety: What, Why, and How", *ACM Computing Surveys*, Vol. 18, No. 2, June 1986, pp. 125-163.

[12] John D. Musa and A. Frank Ackerman, "Quantifying Software Validation: When to Stop Testing?", *IEEE Software*, Vol. 6. No. 3, May 1989, pp. 19-27.

[13] John D. Musa, et al, *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, New York, 1987.

[14] Norman F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data", *IEEE Transactions on Software Engineering*, Vol. 19, No. 11, November 1993, pp. 1095-1104.

[15] Norman F. Schneidewind and T. W. Keller, "Application of Reliability Models to the Space Shuttle", *IEEE Software*, Vol. 9. No. 4, July 1992 pp. 28-33.

[16] Norman F. Schneidewind, "Analysis of Error Processes in Computer Software", *Proceedings of the International Conference on Reliable Software*, IEEE Computer Society, 21-23 April 1975, pp. 337-346.

[17] Nozer D. Singpurwalla, "Determining an Optimal Time Interval for Testing and Debugging Software", *IEEE Transactions on Software Engineering*. Vol. 17, No. 4, April 1991, pp. 313-319.

[18] Jeffrey M. Voas and Keith W. Miller, "Software Testability: The New Verification", *IEEE Software*, Vol. 12, No. 3, May 1995, pp. 17-28.

[19] *Webster's New Universal Unabridged Dictionary*, Second Edition, Simon and Shuster, New York, 1979.

[20] Elaine J. Weyuker, "Using the Consequences of Failures for Testing and Reliability Assessment", *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington. D.C., October 10-13, 1995, pp. 81-91.

Software quality control and prediction model for maintenance

Norman F. Schneidewind

*Division of Computer and Information Sciences and Operations, Naval Postgraduate School,
2822 Racoon Trail, Pebble Beach, CA 93953, USA*
E-mail: nschneid@nps.navy.mil

We develop a quality control and prediction model for improving the quality of software delivered by development to maintenance. This model identifies modules that require priority attention during development and maintenance by using Boolean discriminant functions. The model also predicts during development the quality that will be delivered to maintenance by using both point and confidence interval estimates of quality. We show that it is important to perform a marginal analysis when making a decision about how many metrics to include in a discriminant function. If many metrics are added at once, the contribution of individual metrics is obscured. Also, the marginal analysis provides an effective rule for deciding when to stop adding metrics. We also show that certain metrics are dominant in their effects on classifying quality and that additional metrics are not needed to increase the accuracy of classification. Related to this property of *dominance* is the property of *concordance*, which is the degree to which a set of metrics produces the same result in classifying software quality. A high value of *concordance* implies that additional metrics will not make a significant contribution to accurately classifying quality; hence, these metrics are redundant. Data from the *Space Shuttle* flight software are used to illustrate the model process.

1. Introduction

A key problem in maintenance is to identify problems in the software during development before it reaches maintenance. To this end, we develop a quality control and prediction model that is used to identify modules that require priority attention during development and maintenance. This is accomplished in two activities: *validation* and *application*. Both activities occur during software development. *Validation* is an activity that is required in order to identify metrics that can identify low quality software that requires corrective action. *Application* is an activity during which validated metrics are applied to control and predict software quality. During *validation*, we use a build of the software that has been developed as the source of data to compute a discriminant function (i.e., a statistical method that is used to classify software quality) that we use to retrospectively classify and predict quality with specified accuracy, by build and module. Using this discriminant function during *application*, we classify and predict the quality of new software that is being developed. We make both point and

confidence interval estimates of quality. This is the quality we expect to experience during maintenance.

During *validation*, both quality factor (e.g., discrepancy reports of deviations between requirements and implementation) and software metrics (e.g., size, structural) data are available; during *application*, only the latter are available. During *validation*, we construct Boolean discriminant functions (BDFs) comprised of a set of metrics and their critical values (i.e., thresholds). A BDF is a Boolean function consisting of *AND* and *OR* operators, module metric values, and metric critical values that is used to classify the quality of software. A metric critical value is a value in the range of the metric, estimated by using the inverse of the *Kolmogorov–Smirnov distance* (to be explained) that provides a threshold between two levels (e.g., *high* and *low*) of the quality of the software. We select the best BDF based on its ability to achieve the maximum relative incremental quality/cost ratio. During *application*, if at least one of the module's metrics has a value that exceeds its critical value, the module is identified as "high priority" (i.e., low quality); otherwise, it is identified as "low priority" (i.e., high quality). Our objective is to identify and correct quality problems during development so that a high quality product can be delivered to maintenance, as opposed to waiting until maintenance when the cost of correction would be high.

We use nonparametric statistical methods to: (1) identify the critical values of the metrics and (2) find the optimal BDF based on its ability to satisfy both *statistical* and *application* criteria. Statistical criteria refer to the ability to correctly classify the software (i.e., classify high quality software as high quality and low quality software as low quality). Application criteria refer to the ability to achieve a high quality/cost ratio. A BDF compares a module's metric value with the metric's critical value, for a set of metrics, in classifying the quality of the software. The BDFs provide good accuracy (i.e., $\leqslant 3\%$ error) for classifying quality factors. These functions make fewer mistakes in classifying software that is low quality than is the case when linear vectors of metrics are used because the critical values provide additional information for discriminating quality. In addition, we develop an effective stopping rule for adding metrics to the BDF that is based on quality/cost considerations.

We show that it is important to perform a marginal analysis (i.e., identification of the incremental contribution of each metric to improving quality) when making a decision about how many metrics to include in the discriminant function. If many metrics are added to the set at once, the contribution of individual metrics is obscured. Also, the marginal analysis provides an effective rule for deciding when to stop adding metrics. We also show that certain metrics are dominant in their effects on classifying quality for *Space Shuttle* software (i.e., dominant metrics make fewer mistakes in classifying metrics than non-dominant ones) and that additional metrics are not needed to accurately classify quality. Related to the property of *dominance* is the property of *concordance*, which is the degree to which a set of metrics produces the same result in classifying software quality. A high value of *concordance* implies that additional metrics will not make a significant contribution to accurately classifying quality; hence, these metrics are redundant.

The contributions of this research are the following:

(1) both statistical and application criteria should be used to determine which metrics and how many metrics should be used to classify maintenance quality;

(2) a marginal analysis should be performed on each metric to determine whether its addition will increase the quality/cost ratio;

(3) the Boolean discriminant function (BDF) is a new type of discriminant for classifying maintenance quality;

(4) our application of the Kolmogorov–Smirnov (K–S) distance is a new way to determine a metric's critical value; and

(5) we have developed a new stopping rule for adding metrics: the ratio of the relative improvement in quality to the relative increase in cost.

1.1. Related research

Our model is one of a class of models concerned with the classification of quality, sometimes referred to as the identification of fault-prone modules. Porter and Selby [1990] used classification trees to partition multiple metric value space so that a sequence of metrics and their critical values could be identified that were associated with either high quality or low quality software. This technique is closely related to our approach of identifying a set of metrics and their critical values that will satisfy quality and cost criteria. However, we use statistical analysis to make the identification.

Briand et al. [1998] used logistic regression to classify modules as fault-prone or not fault-prone as a function of various object oriented metrics. In another example of logistic regression, Khoshgoftaar and Allen [1997] used it to classify modules as fault-prone or not fault-prone as a function of faults, requirements, performance, and documentation software trouble report metrics. While one of our objectives is similar – classify modules as either high quality or low quality – we derive from this *binary* classification several predictive *continuous* quality and cost metrics. These metrics are used to predict the quality of software that will be delivered by development to maintenance and the cost of achieving it.

Khoshgoftaar *et al.* [1996a] used nonparametric discriminant analysis in each iteration of their military system project to predict fault-prone modules in the next iteration. This approach provided an advance indication of reliability and the risk of implementing the next iteration. They also conducted a similar study involving a telecommunications application, again using nonparametric discriminant analysis, to classify modules as either fault-prone or not fault-prone [Khoshgoftaar *et al.* 1996b]. Our approach has the same objective but we produce BDFs in terms of the original metrics as opposed to using density functions as discriminators.

Khoshgoftaar and Allen [1998] have also developed models for ranking modules for reliability improvement according to their degree of fault-proneness as opposed to whether they are fault-prone or not. They used Alberg Diagrams [Ohlsson and

Alberg 1996] that predict percentage of faults as a function of percentage of modules by ordering modules in decreasing order of faults and noting the cumulative number of faults corresponding to various percentages of modules. The imperative in safety critical systems like the *Space Shuttle* is to investigate *all* suspect modules because even the module with the lowest a priori reliability risk could pose a safety hazard in operation. Our previous research showed a very high association between module failures and metric values that exceeded the critical values [Schneidewind 1995], as we will show later.

The following topics are covered: *Discriminative Power* model, approach to validation, and quality control and prediction applications of the model, section 2; detailed description of validation methodology, section 3; comparison of validation with application results for quality control and prediction, section 4; quality point and confidence interval estimates, section 5; comparison of BDF and linear discriminant function quality classification results, section 6; development metric characteristics of modules that failed during maintenance. section 7; and conclusions about the contributions of the model to quality control and prediction and the results obtained to date in applying it to the *Space Shuttle*, section 8.

2. Discriminative power model

2.1. Discriminative power validation

Using our metrics validation methodology [IEEE 1998; Schneidewind 1992], and the *Space Shuttle* flight software metrics and discrepancy reports (DRs). we validate metrics with respect to the quality factor *drcount*. This is the number of discrepancy reports written against a module. In brief, this involves conducting statistical tests to determine whether there is a high degree of association between *drcount* and candidate metrics. As shown in figure 1, we validate metrics on one random sample (validation sample) of 100 modules from Build 1 and apply the validated metrics to three random samples (application samples) of 100 modules each from Build 2 that are both disjoint among themselves and from the validation sample, drawn from a population of 1397 modules of *Space Shuttle* flight software. Nikora and Munson argue for the need of a measurement baseline against which evolving systems may be compared [Nikora and Munson 1998]. Our baseline is Build 1 in figure 1. The measurement results from Build 1 provide the data source for controlling and predicting the quality delivered to maintenance and for comparing predicted with actual quality, once the latter is known. Next, we define *Discriminative Power*.

2.1.1. Discriminative Power
Given the elements M_{ij} of a matrix of n modules and m metrics (i.e.. nm metric values), the elements MC_j of a vector of m metric critical values, the elements F_i of a vector of n quality factor values, and scalar FC of quality factor critical value, M_{ij}

M_{ij} : Metric j on Module i
MC_j: Metric j Critical Value
F_i : Quality Factor on Module i
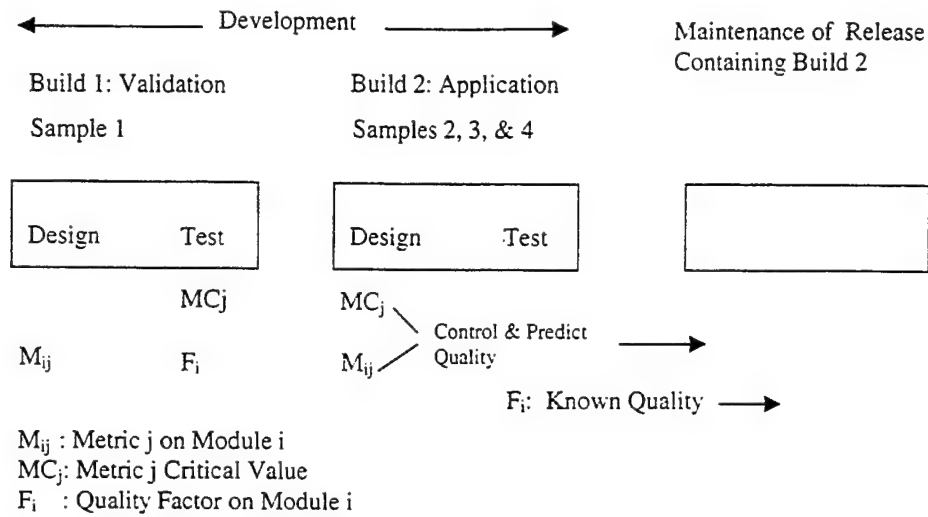
Figure 1. Measurement process.

must be able to discriminate with respect to F_i, for a specified FC, as shown in the following relation:

$$M_{ij} > MC_j \Leftrightarrow F_i > FC \quad \text{and}$$
$$M_{ij} \leqslant MC_j \Leftrightarrow F_i \leqslant FC \tag{1}$$

for $i = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, m$ with specified α, where α is the significance level of various statistical tests that are used for estimating the degree to which a set of metrics can correctly classify software quality. In other words, do the indicated metric relations imply corresponding quality factor relations in (1)? This criterion assesses whether MC_j has sufficient *Discriminative Power* to be capable of distinguishing a set of high quality modules from a set of low quality modules. If this is the case, we use the critical values in Quality Control and Prediction described below. The validation process is illustrated in figure 1, where the critical values MC_j are produced in the Test phase of Build 1 by using the metrics M_{ij} from the Design phase and the quality factor F_i (e.g., *drcount*) that is available in the Test phase. Discrepancy reports are written against the software throughout development but they are not significantly complete until the end of the Test phase for a build during which failures are observed. The counts of discrepancy reports and metrics that are associated with a module were collected at the completion of a build by a metrics analyzer, using the source code as input. If a discrepancy report involves multiple modules, it is counted against every module affected. The desired quality level is set by the choice of FC. The lower its value, the higher the quality requirement; conversely, the higher its value, the lower the requirement. A value of zero is appropriate for safety-critical systems like the *Space Shuttle*.

It is important to recognize that validation is performed retrospectively. That is, with both metrics M_{ij} and quality factor F_i in hand for Build 1, we can evaluate how well the metrics would have performed if they had been applied to Build 1. If the

metrics perform well, we say they are validated and it is our expectation that they will perform adequately when applied to Build 2. (i.e., not as well as when applied to Build 1 because of possible differences in module characteristics between Build 1 and Build 2 but better than using unvalidated metrics). Next, we describe the application of the model to quality control and prediction.

2.1.2. Quality control and prediction

Quality control is the evaluation of modules with respect to predetermined critical values of metrics. The purpose of quality control is to allow software managers to identify software that does not meet quality requirements early in the development process so corrective action can be taken when the cost is low. Quality control is applied during the Design phase of Build 2 in figure 1 to flag modules below quality limits for detailed inspection. The validated BDFs, comprised of the metrics M_{ij} and their critical values MC_j that are obtained from Build 1, are used to either accept or reject the modules of Build 2 [Schneidewind 1997a,b]. At this point in the development of Build 2, only the metric data M_{ij} and MC_j are available.

Quality predictions are used by the developer and maintainer to anticipate rather than react to quality problems. The predictions provide indications of the quality of the software that would be delivered to maintenance. Figure 1 shows the metrics controlling and predicting the quality of software that will be delivered to maintenance *early* in the development of Build 2. Accompanied by rigorous inspection and test, this process will result in improved quality of Build 2 and the software that is released to maintenance. of which Build 2 is a part. Once all of the quality factor data F_i (e.g., *drcount*) have been collected for Build 2, at the end of the Test phase as shown in figure 1, the quality of Build 2 would be known. This, then, becomes the actual quality of Build 2 in the maintained software.

3. Validation methodology

The basis of this model is a methodology for validating BDFs and their critical values that have the ability to discriminate high quality from low quality. We use a three-stage process for selecting metrics for quality control and prediction:

(1) compute critical values of the candidate metrics;

(2) for the set of candidate metrics and critical values, find the optimal combination based on statistical and application criteria; and

(3) apply a stopping rule for adding metrics.

Table 1 provides a functional description of each stage. The three stages take place during the Test phase of Build 1 of figure 1, once all the quality factor data F_i (e.g., *drcount*) are available. The sections that follow provide the details of the statistical analysis for each stage.

Table 1
Functional description of metrics validation process.

	Statistical test/procedure	Purpose	Result
Stage 1	Kolmogorov–Smirnov (K–S)	Compute the critical values of the candidate metrics.	Metrics ranked by K–S test results for input to stage 2.
Stage 2	Contingency table analysis	Use the critical values obtained from stage 1 to form a set of BDFs. Use the BDFs to estimate quality and cost of inspection for each set of metrics, starting with one metric, and increasing by one until the stopping rule is satisfied.	Metric sets with increasing numbers of metrics, each set with estimated quality and cost of inspection.
Stage 3	Stopping rule for adding metrics	Add metrics to stage 2 until the ratio of relative incremental quality to relative incremental inspection cost reaches a maximum.	Validated BDFs and their critical values that provide the highest estimated quality relative to the estimated cost of inspection.

Table 2
Kolmogorov–Smirnov distance for *drcount* = 0 vs. *drcount* > 0. Validation sample 1 (n = 100 modules).

Metric (symbol)	Definition (counts per module)	Critical value	Distance	α	Rank
Prologue size (P)	Change history line count in module listing	38	0.585	0.005	1
Statements (S)	Executable statement count	26	0.557	0.005	2
Eta1 ($E1$)	Unique operator count	10	0.492	0.005	3
Nodes (N)	Node count (in control graph)	11	0.487	0.005	4

3.1. Stage 1: compute critical values

Critical values MC_j are computed, using a new method we have developed, which is based on the Kolmogorov–Smirnov (K–S) test [Conover 1971]. This test was investigated for application to software metrics because of its ability to indicate the value of a metric (i.e., critical value) where maximum discrimination occurs between two samples of modules – one of high quality and the other of low quality. The method has consistently yielded good results for controlling the quality of *Space Shuttle* software as our results will show. The K–S test is exact for continuous distributions and conservative (i.e., the true alpha is less than the specified value) for discrete metrics data [Conover 1971]. In addition, the large range (e.g., 0–2316 for *prologue size*) and fine granularity (e.g., units of one for *prologue size*) of the metrics data approximate continuous distributions. Thus, the K–S test is appropriate for analyzing metrics data.

Table 2 shows the metric definitions, critical values MC_j, and K–S distances for four metrics of the validation sample. These metrics were selected for analysis based on their relatively high K–S distance compared to other metrics that had been
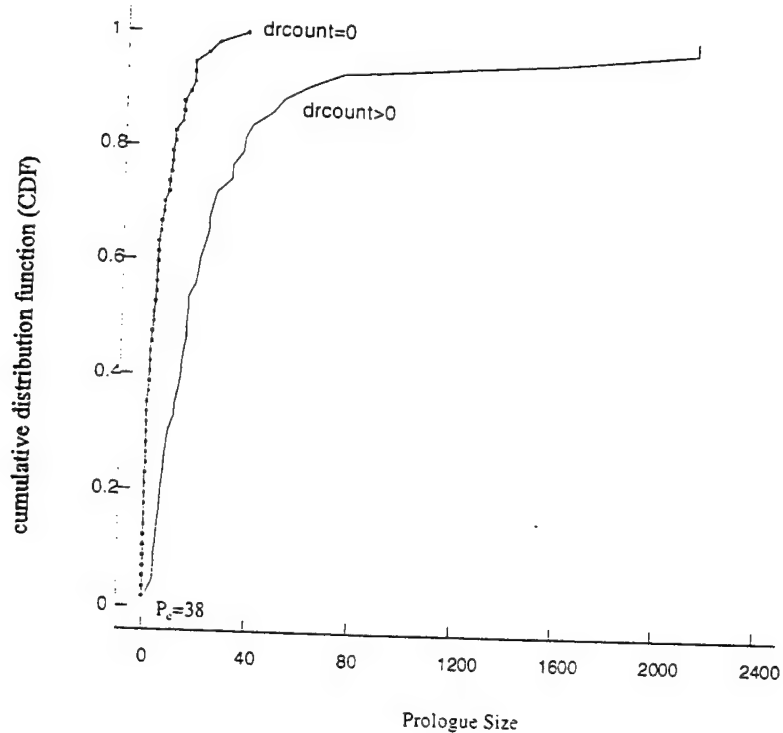
Figure 2. K–S test: Prologue size CDF (sample 1, $n = 100$ modules).

collected on the *Space Shuttle*. The K–S method tests whether the sample cumulative distribution functions (CDF) are from the same or different populations. The test statistic is the maximum vertical difference between the CDFs of two samples (e.g., the CDFs of M_{ij} for *drcount* $\leqslant$ FC and *drcount* $>$ FC). If the difference is significant (i.e., $\alpha \leqslant 0.005$), the value of M_{ij} corresponding to maximum CDF difference is used for MC_j. This relationship is expressed in equation (2). This concept is illustrated in figure 2, for the critical value of *prologue size*, where we show the CDFs for *drcount* $= 0$ and *drcount* > 0. In this example, the critical value is 38. This is the value of *prologue size* where there is the maximum difference between the CDFs. This is the value of *prologue size* where there is the maximum discrimination between high quality (*drcount* $= 0$ curve) and low quality (*drcount* > 0 curve). Metrics are added to the BDF in the order of their decreasing K–S distance:

$$K\text{–}S(MC_j) = \max\left\{\left[CDF(M_{ij} \mid F_i \leqslant FC)\right] - \left[CDF(M_{ij} \mid F_i > FC)\right]\right\}. \quad (2)$$

The history of changes (e.g., requirements, design, and code) and other activities (e.g., inspections, tests, and failure and fault observations) are recorded at the beginning of a module's listing (i.e., *prologue*). The number of lines in this section is called the *prologue size*. Because this metric records the volatility of the software, it is a very good quality discriminator, as our results will demonstrate. A *statement* is an

executable statement in the Hal/S programming language that is used to code the *Space Shuttle* flight software.

3.2. Stage 2: perform contingency table analysis

3.2.1. Validation contingency table

For each BDF identified in stage 1 we use the *contingency table* (see table 3) and its accompanying χ^2 statistic [Conover 1971] to further evaluate the ability of the functions to discriminate high quality from low quality, from both statistical (e.g., values of χ^2 and α) and application (e.g., ability of the metric set to correctly classify low quality modules) standpoints. In table 3, MC_j and FC classify modules into one of four categories. The left column contains modules where none of the metrics exceeds its critical value; this condition is expressed with a Boolean AND function of the metrics. This is the *ACCEPT* column, meaning that according to the classification decision made by the metrics, these modules have acceptable quality. The right column contains modules where at least one metric exceeds its critical value; this condition is expressed by a Boolean *OR* function of the metrics. This is the *REJECT* column, meaning that according to the classification decision made by the metrics, these modules have unacceptable quality. The top row contains modules that are high quality; these modules have a quality factor that does not exceed its critical value (e.g., *drcount* = 0). The bottom row contains modules that are low quality; these modules have a quality factor that exceeds its critical value (e.g., *drcount* > 0).

Equation (3) gives the algorithms for making the cell counts of modules, using the BDFs of F_i and M_{ij} that are computed over the n modules for m metrics. This equation is an implementation of the relation given in (1).

$$C_{11} = \underset{i=1}{\overset{n}{\text{COUNT}}} \text{ FOR}\big((F_i \leqslant \text{FC}) \wedge (M_{i1} \leqslant \text{MC}_1) \wedge \cdots \wedge (M_{im} \leqslant \text{MC}_m)\big),$$

$$C_{12} = \underset{i=1}{\overset{n}{\text{COUNT}}} \text{ FOR}\big((F_i \leqslant \text{FC}) \wedge (M_{i1} > \text{MC}_1) \vee \cdots \vee (M_{im} > \text{MC}_m)\big),$$

$$C_{21} = \underset{i=1}{\overset{n}{\text{COUNT}}} \text{ FOR}\big((F_i > \text{FC}) \wedge (M_{i1} \leqslant \text{MC}_1) \wedge \cdots \wedge (M_{im} \leqslant \text{MC}_m)\big), \tag{3}$$

$$C_{22} = \underset{i=1}{\overset{n}{\text{COUNT}}} \text{ FOR}\big((F_i > \text{FC}) \wedge (M_{i1} > \text{MC}_1) \vee \cdots \vee (M_{im} > \text{MC}_m)\big),$$

for $j = 1, \ldots, m$, and where

$$\text{COUNT}(i) = \begin{cases} \text{COUNT}(i-1) + 1 & \text{FOR Boolean expression } true, \\ \text{COUNT}(i-1) & \text{otherwise;} \end{cases}$$

$$\text{COUNT}(0) = 0.$$

The counts correspond to the cells of the *contingency table* (C_{11}, C_{12}, C_{21}, and C_{22}), as shown in table 3, where row and column totals are also shown: n, n_1, n_2, N_1, and N_2. The analysis could be generalized to include multiple quality factors, if necessary; in this case, the *contingency table* would have more than two rows.

Table 3
Validation contingency table.

	$\bigwedge(M_{ij} \leqslant MC_j)$ $P_i \leqslant 38 \wedge S_i \leqslant 26$	$\bigvee(M_{ij} > MC_j)$ $P_i > 38 \vee S_i > 26$	
High quality $F_i \leqslant FC$ $drcount = 0$	$C_{11} = 30$	$C_{12} = 27$ type 2	$n_1 = 57$
Low quality $F_i > FC$ $drcount > 0$	$C_{21} = 1$ type 1	$C_{22} = 42$	$n_2 = 43$
	$N_1 = 31$ RF = 1, RFM = 1	$N_2 = 69$	$n = 100$ TF = 192
	ACCEPT	REJECT	

In addition to counting modules in table 3, we must also count the quality factor (e.g., *drcount*) that is incorrectly classified. This is shown as Remaining Factor, RF, in the *ACCEPT* column. This is the quality factor count on modules that should have been rejected. Also shown is Total Factor. TF, the total quality factor count on all the modules in the sample (i.e., the sum of *drcount*). Lastly we show RFM (Remaining Factor Modules) that is the count of modules with quality factor count >0 (i.e., modules with Remaining Factor, RF).

Table 3 and subsequent equations show an example validation, where the optimal combination of metrics from table 2 and their critical values for a random sample of 100 modules (sample 1), from the population of 1397, is *prologue size* (P) with a critical value of 38 and *statements* (S) with a critical value of 26. This low value of *statements* is understandable because the median value in the builds analyzed is 23. There are many small modules that call a subroutine, compute a value, and transfer control to another module. Later we will explain how we arrived at this particular combination of metrics as the optimal set.

3.2.2. Statistical criteria

We validate a BDF statistically by demonstrating that it partitions table 3 in such a way that C_{11} and C_{22} are large relative to C_{12} and C_{21}. If this is the case, a large number of high quality modules (e.g., modules with $drcount = 0$) would have $M_{ij} \leqslant MC_j$ and would be correctly classified as high quality. Similarly, a large number of low quality modules (e.g., modules with $drcount > 0$) would have $M_{ij} > MC_j$ and would be correctly classified as low quality. One measure of the degree to which this is the case is estimated by the chi-square (χ^2) statistic [Conover 1971]. If computed $\chi_c^2 > \chi_s^2$ (chi-square at specified α_s) and if computed $\alpha_c < \alpha_s$, then these results suggest that a given BDF can discriminate between high and low quality. However, because the χ^2 test may not produce consistent results [Eman 1998], we use it only as *one* of several indicators of *Discriminative Power*. Other criteria are misclassification rates and, most important, application criteria (see below). We note that the use of

chi-square and alpha as statistical criteria is independent of the application (i.e., these criteria could be used whether the application is metrics or personnel management). Application criteria, on the other hand, such as *quality* and *inspection* (see below) are meaningful in the *context* of the metrics application.

3.2.2.1. Misclassification

We compute the degree of misclassification in table 3 by noting that ideally $C_{11} = n_1 = N_1$, $C_{12} = 0$, $C_{21} = 0$, $C_{22} = n_2 = N_2$. The extent that this is not the case is estimated by *type 1* misclassifications (i.e., the module has *low quality* and the metrics "say" it has *high quality*) and *type 2* misclassifications (i.e., the module has *high quality* and the metrics "say" it has *low quality*). Thus, we define the following measures of misclassification:

Proportion of modules of type 1: $$P_1 = \frac{C_{21}}{n}. \tag{4}$$

Proportion of modules of type 2: $$P_2 = \frac{C_{12}}{n}. \tag{5}$$

Proportion of modules of type 1 + type 2: $$P_{12} = \frac{C_{21} + C_{12}}{n}. \tag{6}$$

For the example, $P_1 = (1/100) \cdot 100 = 1\%$, $P_2 = (27/100) \cdot 100 = 27\%$, $P_{12} = ((1 + 27)/100) \times 100 = 28\%$.

3.2.3. Application criteria

It is insufficient to validate only with respect to statistical criteria. In the final analysis, it is the performance of the metrics in the application context that counts. Therefore, we validate metrics with respect to the application criteria: *quality and inspection*, which are related to the quality achieved and the cost to achieve it, respectively [Schneidewind 1997a,b]. At the Design phase of Build 2 in figure 1, we predict that the quality computed by equations (7)–(12) will be delivered to maintenance, assuming that the modules that are rejected by the quality control process are inspected and tested and that the problems that are found are corrected. Furthermore, we predict that the degree of inspection computed by equation (13) will be required to achieve this quality.

3.2.3.1. Quality

First, we estimate the ability of the metrics to correctly classify quality, given that the quality is known to be low: proportion of low quality (e.g., *drcount* > 0) modules correctly classified

$$LQC = \frac{C_{22}}{n_2}. \tag{7}$$

For the example, $LQC = (42/43) \cdot 100 = 97.7\%$.

Second, we estimate the ability of the metrics to correctly classify quality, given that the BDF has classified modules as *ACCEPT*. This is done by summing the quality

factor in the *ACCEPT* column in table 3 to produce Remaining Factor, RF (e.g., remaining *drcount*), given by equation (8):

$$RF = \sum_{i=1}^{n} F_i \, \text{FOR}\big((F_i > FC) \wedge (M_{i1} \leqslant MC_1) \wedge \cdots \wedge (M_{ij} \leqslant MC_j) \wedge \cdots$$

$$\wedge (M_{im} \leqslant MC_m)\big), \quad \text{for } j = 1, \ldots, m. \tag{8}$$

This is the sum of quality factor F_i (e.g., *drcount*) on modules incorrectly classified as high quality because $(F_i > FC) \wedge (M_{ij} \leqslant MC_j)$ for these modules. We assume that the elements of F_i are additive and that the lower its value, the higher the quality of the module. This would be the case for any quality factor of interest in this analysis: discrepancy report count, error count, fault count, and failure count.

We estimate the proportion of RF by equation (9), where TF is the total quality factor F_i for the validation sample:

$$RFP = \frac{RF}{TF}. \tag{9}$$

For the example. from table 3 there is a one DR on one module that is incorrectly classified (i.e., RF = 1). The total number of DRs for the 100 modules is 192. Therefore, RFP = (1/192) · 100 = 0.52%.

We estimate the density of RF by equation (10):

$$RFD = \frac{RF}{n}. \tag{10}$$

For the example, RFD = 1/100 = 0.01 *drcount*/module.

In addition, we estimate the count of modules that were incorrectly classified because they have DRs written against them (i.e., have $F_i > FC$). The proportion remaining RMP is given by equation (11). Note that RMP = P_1 (proportion of *type 1* misclassifications) when FC = 0 (i.e., the only modules with $F_i > 0$ will be in the C_{21} cell); see table 3.

$$RMP = \frac{RFM}{n}, \tag{11}$$

where RFM is given by

$$RFM = \underset{i=1}{\overset{n}{\text{COUNT}}} \, \text{FOR}\big((F_i > 0) \wedge (M_{i1} \leqslant MC_1) \wedge \cdots \wedge (M_{ij} \leqslant MC_j)$$

$$\wedge \cdots \wedge (M_{im} \leqslant MC_m)\big), \quad \text{for } j = 1, \ldots, m. \tag{12}$$

For the example, there is one accepted module with one DR, so RMP = (1/100)·100 = 1%.

3.2.3.2. Inspection

Inspection is one of the costs of high quality. We are interested in weighing inspection requirements (i.e.. percent of modules rejected and subjected to detailed

Table 4
Discriminative Power validity evaluation (sample 1, $n = 100$ modules).

Metric set	Critical values				Statistical criteria				Application criteria			
	P	S	$E1$	N	P_1 $\%c$	P_2 $\%c$	χ^2_c	α_c for χ^2_c	LQC $\%c$	RFP $\%c$	RMP $\%c$	I $\%c$
P	38				2	21	33.2	8.4×10^{-9}	95.3	1.56	2	62
P,S	38	26			1	27	26.7	2.4×10^{-7}	97.7	0.52	1	69
$P,S,E1$	38	26	10		1	30	22.5	2.1×10^{-6}	97.7	0.52	1	72
K–S distance	0.585	0.557	0.492	0.487								

P: prologue size, S: statements. $E1$: etal, N: nodes

inspection) against the quality that is achieved, for various BDFs. We estimate inspection requirements by noting that all modules in the *REJECT* column of table 3 must be inspected; this is the count $C_{12} + C_{22}$. Thus, the proportion of modules that must be inspected is given by

$$I = \frac{C_{12} + C_{22}}{n}. \tag{13}$$

For the example, $I = ((27 + 42)/100) \cdot 100 = 69\%$ and the percentage accepted is $1 - I = 31\%$.

3.2.4. Summary of validation results

The results of the validation example are summarized in table 4. The properties of *dominance* and *concordance* are evident in these validation results and in other samples we have analyzed from this data. That is, a point is reached in adding metrics where *Discriminative Power* is not increased because: (1) the contribution of the dominant metrics in correctly classifying quality has already taken effect, and (2) additional metrics essentially replicate the classification results of the dominant metrics – the *concordance* effect. This result is due to the property of the BDF used as an OR function, which will cause a module to be rejected if only one of the module's metrics exceeds its critical value. These effects can only be observed if a marginal analysis is performed, where metrics are added to the set one-by-one and the calculations shown in table 4 are made after each metric is added. For each added metric, its effect is evaluated with respect to both statistical and application criteria. In addition, a suitable stopping rule must be used to know when to stop adding metrics (see the next section).

3.3. Stage 3: Apply a stopping rule for adding metrics

One rule for stopping the addition of metrics to a BDF is to quit when RFP no longer decreases as metrics are added. This is the *maximum quality* rule. This rule is illustrated in table 4. When a third metric, *etal* ($E1$), is added, there is no decrease in RFP and RMP nor is there an increase in LQC. If it is important to strike a balance between quality and cost (i.e., between RFP and I), we add metrics until the ratio of the relative change in RFP to the relative change in I is maximum, as given by the

Table 5
Application contingency table.

	$\bigwedge(M_{ij} \leqslant MC_j)$ $P_i \leqslant 38 \wedge S_i \leqslant 26$	$\bigvee(M_{ij} > MC_j)$ $P_i > 38 \vee S_i > 26$	
High quality ?	?	Type 2 ?	?
Low quality ?	Type 1 ?	?	?
	$N_1 = 40$	$N_2 = 60$	$n = 100$
	ACCEPT	REJECT	

Quality Inspection Ratio (QIR) in equation (14), where i refers to the previous RFP and I:

$$QIR = \frac{|\Delta RFP|/RFP_i}{\Delta I/I_i}. \tag{14}$$

For the example,

$$QIR(P \to P, S) = \frac{|0.52 - 1.56|/1.56}{(69 - 62)/62} = 5.90.$$

This is the value of QIR in going from one metric *prologue size* (P) to two metrics (P, S), adding *statements* (S).

Also, $QIR(P, S \to P, S, E1) = 0$. This is the value of QIR in going from two metrics (P, S) to three metrics $(P, S, E1)$, adding *etal* ($E1$).

Therefore, we stop adding metrics after *statements* has been added. In this particular case, equation (14) produces the same metric set as the *maximum quality* rule.

4. Comparison of validation with application results

In order to compare validation with application results, we first show how the Contingency table looks at the Design phase of Build 2 in figure 1, when only the metrics M_{ij} and their critical values MC_j are available. This is shown in table 5, where the "?" indicates that the quality factor data F_i are not available when the validated metrics are used in the quality control function of Build 2. During the Design phase of Build 2, modules are classified according to the criteria that have been described. A second disjoint random sample of 100 modules (sample 2) was used to illustrate the process. Whereas 31 and 69 modules were accepted and rejected, respectively, during Build 1, 40 and 60 modules were accepted and rejected, respectively, during Build 2. The rejected modules would be given priority attention (i.e., subjected to rigorous inspection).

A comparison of the validation sample (Build 1) with the application samples (Build 2) with respect to statistical criteria is shown in table 6. A comparison of the

Table 6

Statistical criteria P1 and P2 for metric set: P, S. Validation (sample 1) vs. application (samples 2–4), $n = 100$ modules.

P1: percentage type 1 misclassification				P2: percentage type 2 misclassification			
Sample 1	Sample 2	Sample 3	Sample 4	Sample 1	Sample 2	Sample 3	Sample 4
1.0	1.0	4.0	3.0	27.0	24.0	18.0	22.0

Table 7

Application criteria LQC and RFP for metric set: P, S. Validation (sample 1) vs. application (samples 2–4), $n = 100$ modules.

LQC: percentage of low quality modules (*drcount* > 0) correctly classified				RFP: percentage of quality factor (*drcount*) incorrectly classified			
Sample 1	Sample 2	Sample 3	Sample 4	Sample 1	Sample 2	Sample 3	Sample 4
97.7	97.3	91.1	93.2	0.52	0.62	3.01	1.50

Table 8

Application criteria RFD and I for metric set: P, S. Validation (sample 1) vs. application (samples 2–4), $n = 100$ modules.

RFD: density of quality factor (*drcount*/module) incorrectly classified				I: percentage of modules inspected			
Sample 1	Sample 2	Sample 3	Sample 4	Sample 1	Sample 2	Sample 3	Sample 4
0.01	0.01	0.05	0.03	69	60	59	63

validation sample with the application samples with respect to application criteria is shown in tables 7 and 8. As we have mentioned, only metrics data is available when the validated metrics are applied during the Design phase of Build 2 in figure 1. However, to have a basis for comparison with the validation results, we computed the values shown in tables 6–8 *retrospectively* (i.e., after Build 2 was far enough along to be able to collect all of the quality factor data at the conclusion of the Test phase). The values for samples 2–4 in tables 7 and 8 are the *actual* quality delivered to maintenance, as shown during the Test phase of figure 1. The reader should compare the results of samples 2–4 with those of sample 1 in the tables. As the accuracy of classification of low quality software increases, the accuracy of classifying high quality software decreases and inspection cost increases. However, the more important consideration is to prevent low quality software from being delivered to maintenance, particularly in safety critical systems like the *Space Shuttle*.

5. Quality point and confidence interval estimates

In addition to the quantities in tables 3–8, there are other quantities of interest, such as the proportion of modules with zero and non-zero *drcount* and their confidence

intervals. For these quantities, software developers and maintainers are provided with both point estimates and interval estimates of the range in which the actual quality values are likely to fall. Thus, they are able to anticipate rather than react to quality problems. For example, estimates obtained from Build 1 in figure 1 are used to predict the quality of software that would be delivered to maintenance if corrective action were not taken. This action is the quality control step of the Design phase of Build 2 where modules are rejected and subjected to detailed inspection and test if their metrics values exceed the critical values. In addition, the estimates provide indications of resource levels that are needed to achieve quality goals. For example, if the predicted quality of the software were lower than the specified quality, the difference would be an indication of increased usage of personnel and computer time during inspection and testing, respectively.

A benefit of using confidence limits is that they provide protection against prediction error. A prediction error could arise because the very act of measuring and predicting may affect the predictions – the *Heisenberg Principle*. For example, *prologue size*, the record of change history, has proven to be a good predictor of quality. However, if the software is changed in response to problems observed during the quality control function, thereby adding to the change history and *prologue size*, this effect would tend to make the original predictions optimistic. Another protection against prediction error is to periodically repeat the predictions as the software evolves over the life cycle.

The normal approximation to the binomial distribution is used to estimate the confidence limits of the proportions. This distribution is used because we are interested in estimating the proportions of modules and *drcount* that fall into one of two categories (i.e., a module is either accepted or rejected or DRs are either present or not present on a module). The normal approximation gives the mean proportion p of modules or DRs that fall into one of two categories and the confidence limits are a function of p.

The point and confidence limit estimates for module and quality factor counts use terms that are defined below. Where it is necessary to distinguish validation from application quantities in the computations, we use primed notation for the latter.

n: number of modules in the validation and application samples (see tables 3 and 5, respectively).

N_1: number of modules accepted in the validation sample of Build 1.

N_2: number of modules rejected in the validation sample of Build 1.

N'_1: number of modules accepted in the application samples of Build 2.

N'_2: number of modules rejected in the application samples of Build 2.

5.1. Module counts

Module count estimates are made using the validation sample in the Test phase of Build 1. These estimates are applied to the application samples in the Design phase of Build 2 and compared with actual values in table 9.

The proportion of *all* modules with quality factor $F_i > 0$ (e.g., *drcount* > 0 on module i) in the *entire* validation sample is given by equation (15):

$$p_n = \frac{\text{COUNT}_{i=1}^n \text{ FOR } F_i > 0}{n},$$ (15)

where

$$\text{COUNT}(i) = \begin{cases} \text{COUNT}(i-1) + 1 & \text{FOR expression } true, \\ \text{COUNT}(i-1), & \text{otherwise;} \end{cases}$$
$$\text{COUNT}(0) = 0.$$

We use this equation to estimate p_n' in the application samples. We obtain the two-sided confidence interval of p_n from expression (16). We use this expression to estimate the lower and upper limits of p_n' in the application samples:

$$p_n \pm Z_{\alpha/2} \sqrt{\frac{(p_n)(1 - p_n)}{n}}.$$ (16)

As shown in table 9, we would expect the proportion of *all* modules with *drcount* > 0 in maintenance to be between 33.3–52.7% unless corrective action is taken to make these limits lower. If corrective action is taken, this estimate provides bounds on the resources – personnel and computer time – that would be required to inspect, correct, and test defective modules.

The proportion of *accepted* modules with quality factor $F_i > 0$ (e.g., *drcount* > 0 on module i) in the validation sample is given by equation (17), where RFM is obtained from equation (12):

$$pN_1 = \frac{\text{RFM}}{N_1}.$$ (17)

We use this equation to estimate pN_1' in the application samples. We obtain the one-sided upper confidence limit of pN_1 from expression (18). We use this expression to estimate the upper limit of pN_1' in the application samples:

$$pN_1 + Z_\alpha \sqrt{\frac{(pN_1)(1 - pN_1)}{N_1}}.$$ (18)

As shown in table 9, we would expect the proportion of *accepted* modules with *drcount* > 0 in maintenance to be $\leqslant 8.45\%$ as the result of the quality control effort in the Design phase of Build 2.

The proportion of *rejected* modules with quality factor $F_i > 0$ (e.g.. *drcount* > 0 on module i) in the validation sample is given by equation (19):

$$pN_2 = \frac{(p_n)(n) - (\text{RFM})}{N_2}.$$ (19)

This is equal to: (*all* modules with quality factor $F_i > 0$) minus (*accepted* modules with quality factor $F_i > 0$), divided by the number of rejected modules. We use this

equation to estimate pN_2' in the application samples. We obtain the one-sided lower confidence limit of pN_2 from expression (20). We use this expression to estimate the lower limit of pN_2' in the application samples:

$$pN_2 - Z_\alpha \sqrt{\frac{(pN_2)(1 - pN_2)}{N_2}}. \tag{20}$$

As shown in table 9, we would expect the proportion of *rejected* modules with *drcount* > 0 in maintenance to be $\geqslant 51.2\%\%$ as the result of the quality control effort in the Design phase of Build 2.

5.2. Quality factor counts

Quality factor *proportion* count estimates in (21)–(24) are made using the validation sample in the Test phase of Build 1. Quality factor *total* count estimates in (25) and (26) use data from the validation sample and data that is available in the application samples in the Design phase of Build 2: number of modules *accepted*, N_1' and number of modules *rejected*, N_2'. These estimates are applied to the application samples in the Design phase of Build 2 and compared with actual values in tables 9 and 10.

The proportion of quality factor $F_i > 0$ (e.g., *drcount* > 0) that occurs on *accepted* modules in the validation sample is given by equation (21):

$$d_1 = \frac{\text{RF}}{\text{TF}}, \tag{21}$$

where RF is obtained from equation (8) and TF is the total quality factor F_i for the validation sample. We use this equation to estimate d_1' in the application samples. We obtain the one-sided upper confidence limit of d_1 from expression (22). We use this expression to estimate the upper limit of d_1' in the application samples:

$$d_1 + Z_\alpha \sqrt{\frac{(d_1)(1 - d_1)}{\text{TF}}}. \tag{22}$$

As shown in table 9, we would expect the proportion of *drcount* > 0 on *accepted* modules in maintenance to be $\leqslant 1.38\%$ as the result of the quality control effort in the Design phase of Build 2.

The proportion of quality factor $F_i > 0$ (e.g., *drcount* > 0) that occurs on *rejected* modules in the validation sample is given by equation (23):

$$d_2 = 1 - d_1. \tag{23}$$

We use this equation to estimate d_2' in the application samples. We obtain the one-sided lower confidence limit of d_2 from expression (24). We use this expression to estimate the lower limit of d_2' in the application samples:

$$d_2 - Z_\alpha \sqrt{\frac{(d_2)(1 - d_2)}{\text{TF}}}. \tag{24}$$

Table 9
Validation predictions (sample 1) vs. application actual values (samples 2–4).

	Point estimates (sample 1)	95% Confidence limits (sample 1)	Actual values		
			Sample 2	Sample 3	Sample 4
p'_n: proportion of *all* modules with *drcount* > 0	43.0%	33.3–52.7%	37.0%	45.0%	44.0%
pN'_1: proportion of *accepted* modules with *drcount* > 0	3.22%	LE 8.45%	2.50%	9.76%	8.11%
pN'_2: proportion of *rejected* modules with *drcount* > 0	60.9%	GE 51.2%	60.0%	69.5%	65.1%
d_1: proportion of *drcount* > 0 on *accepted* modules	0.52%	LE 1.38%	0.62%	3.01%	1.50%
d'_2: proportion of *drcount* > 0 on *rejected* modules	99.5%	GE 98.6%	99.4%	97.0%	98.5%

As shown in table 9, we would expect the proportion of *drcount* > 0 on *rejected* modules in maintenance to be ⩾ 98.6% as the result of the quality control effort in the Design phase of Build 2.

The total quality factor F_i > 0 (e.g., *drcount* > 0) that occurs on *accepted* modules in the validation sample is given by equation (25):

$$D_1 = \frac{\mathrm{RF}}{N_1} \cdot N'_1.$$ (25)

We use this equation as a predictor of D'_1 in the application samples. As shown in table 10, we would expect the *total drcount* on *accepted* modules in maintenance to be 1.29, 1.32, and 1.19 for application samples 2, 3, and 4, respectively. The reason for the three estimates of sample 1 is that each sample has a different number of *accepted* modules N'_1 in equation (25).

The total quality factor of F_i > 0 (e.g., *drcount* > 0) that occurs on *rejected* modules in the validation sample is given by equation (26):

$$D_2 = \frac{(\mathrm{TF} - \mathrm{RF})}{N_2} \cdot N'_2.$$ (26)

We use this equation as a predictor of D'_2 in the application samples. As shown in table 10, we would expect the *total drcount* on *rejected* modules in maintenance to be 166.1, 163.3, and 174.4 for application samples 2, 3, and 4, respectively. The reason for the three estimates of sample 1 is that each sample has a different number of *rejected* modules N'_2 in equation (26).

Ten of the actual values out of the fifteen cases in table 9 fall within the confidence limits. The average relative error across six comparisons between sample 1 versus

Table 10
Validation actual values and predictions (sample 1) vs. application actual values (samples 2–4).

	Actual sample 1	Estimate sample 1	Actual sample 2	Estimate sample 1	Actual sample 3	Estimate sample 1	Actual sample 4
D_1': *total drcount* on *accepted* modules	1	1.29	1	1.32	5	1.19	3
D_2': *total drcount* on *rejected* modules	191	166.1	160	163.3	161	174.4	197

Table 11
Comparison of Boolean Discriminant Function (BDF) with Linear Discriminant Function (LDF). Validity evaluation (sample 1, $n = 100$ modules).

Function	Metric set	Statistical criteria				Application criteria	
		P_1 (%)	P_2 (%)	χ_c^2	α_c for χ_c^2	LQC (%)	I (%)
BDF	P, S	1.0	27.0	26.7	2.4×10^{-7}	97.7	69.0
LDF	9 metrics	9.0	9.0	37.5	≈ 0	79.1	43.0

LDF metric set (counts per module): Halstead eta1, eta2, $\eta1$, and $\eta2$; lines of code, prologue size, nodes, paths, and maximum path.

samples 2–4 in table 10 is 28.9% with a standard deviation of 30.7%. Variation in results *may* be caused by sampling error (i.e., in order to obtain disjoint samples, it was necessary to sample without replacement).

6. Comparison of Boolean and linear discriminant functions

We compared the quality classifying ability during validation of the Boolean discriminant function (BDF) with an alternate method: the linear discriminant function (LDF) consisting of the summation across metrics of the product of standardized metrics variables and standardized classification coefficients [Jobson 1992]. For the BDF, we used the optimal metrics set – *prologue size* and *statements* – and results obtained from table 4. For the LDF, we used the set of nine metrics listed in table 11 and a marginal analysis that yielded the highest *Discriminative Power* as measured by the eigenvalue and χ^2. The comparison is shown in table 11. In the comparison, we used both statistical and application criteria. In the application category, we did not compute RFP and RMP for the LDF as we did in table 4. Unlike the BDF where equations (8) and (9) count quality factor and (11) and (12) count modules that are misclassified into the *ACCEPT* category, there is no algorithm for making these computations for the LDF. It would have been necessary to compare the metrics and *drcount* for each module with the LDF to determine how the metrics classified the modules and *drcount*. However, a good comparison is obtained by using LQC. In this example, table 10 shows that the BDF does a better job of classifying the low quality modules (e.g., lower value of P_1 and higher value of LQC) and that LDF does a better job of

Table 12
Metric characteristics of failed modules.

Failure number	Severity level	Module ID	Prologue size	Statements	Eta1	Nodes	*drcount*
1	2	13	493	738	46	394	22
2	3	974	299	192	31	98	2
3	2	1286	115	110	28	48	5
4	3	711	205	*1*	*5*	96	6
5	3	1300	82	*3*	*8*	20	1
6	3	515	851	875	44	529	15
7	2	464	69	*15*	16	12	4
7	2	465	76	30	24	21	4
7	2	466	68	*15*	16	12	4
7	2	467	72	30	24	21	2
7	2	468	153	*10*	11	75	3
7	2	472	100	*1*	*6*	40	1
8	4	555	943	819	34	174	26
9	3	904	122	128	31	64	1
10	4	882	157	107	30	51	5
Critical value			38	26	10	11	0
Failed modules mean			253.7	204.9	23.6	110.3	6.7
Build 2 mean			134.6	70.2	16.7	28.4	1.8

classifying the high quality modules (e.g., lower values of P_2 and I). As stated in section 1, the reason for this result is that BDFs make fewer mistakes in classifying software that is low quality than is the case when linear vectors of metrics are used because the critical values provide additional information for discriminating quality. The implications for applying the validated metrics during the quality control function of the Design phase of Build 2 is that the BDF would yield higher quality and the LDF would yield lower cost. Our preference is the BDF in a safety critical system like the *Space Shuttle*, where high quality software is the paramount objective.

7. Metric characteristics of failed modules

Further evidence of the model's ability to identify low quality during development is shown in table 12. This table shows the 15 modules that failed during maintenance of the 1397 modules of Build 2 in figure 1, where the severity of the 10 failures decreases from 2 to 4. In the case of failure #7, six modules caused this failure. The table also shows the module metrics and validated critical values that were obtained during Build 1. For all failed modules, one or more of their metric values exceed the critical value. Metric values in *italics* would fail to reject these modules during quality control of the Design phase of Build 2. However, this would be compensated for by the metric *prologue size* that would have correctly rejected all of these modules. To illustrate the difference in metric characteristics of the failed modules versus all the modules of Build 2, the means of each were computed. The difference in means is

significant at $\alpha < 0.05$. As this example illustrates, although a metrics program can alert the developer to the possibility of unreliable software, it cannot *prevent* failures from occurring. In this example, the inspection and test process failed to find and correct the problems before Build 2 entered maintenance.

8. Conclusions

A model was developed for controlling and predicting the quality of software that is delivered by development to maintenance. The model provides software developers and maintainers with both point estimates and interval estimates of the range in which the actual quality values are likely to fall. Thus, they are alerted to the need to take corrective action.

It is important when validating and applying metrics to consider both statistical and application criteria and to measure the marginal contribution of each metric in satisfying these criteria. When this approach is used, we observe that a point is reached where adding metrics makes no contribution to improving quality and the cost of using additional metrics increases. This phenomenon is due to the metric classification properties of *dominance* and *concordance*. Using our approach, we achieved an error of $\leqslant 3\%$ in classifying quality factors for the samples used in the study. The ratio of the relative improvement in quality to the relative increase in inspection cost is a new and effective stopping rule for adding metrics.

Our Boolean discriminant function (BDF) is a new type of discriminant for classifying software quality to support an *integrated* approach to control and prediction in one model, and our application of the Kolmogorov–Smirnov distance is a new way to determine a metric's critical value. On this application, the BDF, using two metrics, was superior to a linear discriminant function, using nine metrics, in classifying low quality software; however, when used for quality control, the BDF requires more inspection.

Finally, with a very limited sample of modules that caused failures we found that the validated metrics, if they had been applied to the modules that eventually failed, would have acted as early indicators of these failures.

Acknowledgements

References

Briand, L.C., J. Daly, V. Porter, and J. Wüst (1998). "Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems." In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 334–343.

Conover, W.J. (1971), *Practical Nonparametric Statistics*, Wiley, New York.

Eman, K.E. (1998). "The Predictive Validity Criterion for Evaluating Binary Classifiers", In *Proceedings of the Fifth International Metrics Symposium*, IEEE Computer Society Press, Los Alamitos, CA, pp. 235–244.

IEEE Standard for a Software Quality Metrics Methodology (1998), Revision, IEEE Std 1061, IEEE Standards Office, Piscataway, NJ.

Jobson, J.D. (1992), *Applied Multivariate Data Analysis*, Vol. II, Springer-Verlag, New York.

Khoshgoftaar, T.M. and E.B. Allen (1997), "Logistic Regression Modeling of Software Quality," TR-CSE-97-24, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL.

Khoshgoftaar, T.M. and E.B. Allen (1998), "Predicting the Order of Fault-Prone Modules in Legacy Software." In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 344–353.

Khoshgoftaar, T.M., E.B. Allen, R. Halstead, and G. P. Trio (1996a), "Detection of Fault-Prone Software Modules During a Spiral Life Cycle," In *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 69–76.

Khoshgoftaar, T.M., E.B. Allen, K. Kalaichelvan, and N. Goel (1996b), "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software, 13*, 1, 65–71.

Nikora, A.P. and J.C. Munson (1998), "Determining Fault Insertion Rates for Evolving Software Systems," In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 306–315.

Ohlsson, N. and H. Alberg (1996), "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Transactions on Software Engineering, 22*, 12, 886–894.

Porter, A.A. and R.W. Selby (1990), "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software, 7*, 2, 46–54.

Schneidewind, N.F. (1992), "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering, 18*, 5, 410–422.

Schneidewind, N.F. (1995), "Work in Progress Report: Experiment in Including Metrics in a Software Reliability Model," In *Proceedings of the Annual Oregon Workshop on Software Metrics*, Portland State University, Portland, OR, 17 pages.

Schneidewind, N.F. (1997a), "Software Metrics Model for Quality Control," In *Proceedings of the International Metrics Symposium*, IEEE Computer Society Press, Los Alamitos, CA, pp. 127–136.

Schneidewind, N.F. (1997b). "Software Metrics Model for Integrating Quality Control and Prediction," In *Proceedings of the International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 402–415.

The Ruthless Pursuit of the Truth about COTS

Dr. Norman F. Schneidewind
Naval Postgraduate School
2822 Racoon Trail
Pebble Beach
California, 93953, USA
Email: nschneid@nps.navy.mil

Abstract

We expose some of the truths about COTS, discounting some exaggerated claims about the applicability of COTS, particularly with regard to using COTS in safety critical systems. Although we agree that COTS has great potential for reduced development and maintenance time and cost, we feel that the advocates of COTS have not adequately addressed some critical issues concerning reliability, maintainability, availability, requirements risk analysis, and cost. Thus we illuminate these issues, suggesting solutions in cases where solutions are feasible and leaving some questions unanswered because it appears that the questions cannot be answered due to the inherent limitations of COTS. These limitations are present because there is inadequate visibility and documentation of COTS components.

Introduction

In this paper we analyze three important aspects of COTS software: 1) reliability, maintainability, and availability; 2) requirements risk assessment, using risk factors from the Space Shuttle and modifying them for more general use; and 3) cost framework. We are motivated to address these issues because we feel that the COTS community has not adequately addressed some very important questions concerning the applicability of COTS when used in a host system. We define a host system as follows: it contains both COTS and non-COTS software; the latter is specific to the operational mission of the organization; and the mission cannot be satisfied entirely by COTS components. Our concerns are reinforced by Kohl: "The most significant challenges of V&V of COTS products has to do with knowledge of the functionality, performance and quality of these products. Because these products tend to be developed for large,

commercial markets as opposed to being developed to a specification for a single customer, they tend to provide a variety of useful and desirable features for the market that they are targeted for, at the expense of the specific system needs in which such products may be used. Further, quality and reliability are sometimes not considered critical when time-to-market is a driving requirement. Thus, it is sometimes the case that these COTS products contain features and functionality that may not be fully known, even to the vendor." [KOH99].

Many vendors produce products that are not domain specific (e.g., network server) or have limited functionality (e.g., mobile phone). In contrast, many customers of COTS develop systems that are domain specific (e.g., target tracking system) and have great variability in functionality (e.g., corporate information system). This discussion takes the viewpoint of how the customer can ensure the quality of COTS components. In addition to direct quality evaluation, we also consider requirements risk analysis in a later section, which indirectly affects quality. We must distinguish between using a non-mission critical application like a spreadsheet program to produce a budget and a mission critical application like military strategic and tactical operations. Whereas customers will tolerate an occasional bug in the former, zero tolerance is the rule in the latter. We emphasize the latter because this is the arena where there are major unresolved problems in the application of COTS. Furthermore, COTS components may be embedded in host systems. These components must be reliable, maintainable, and available, and must interoperate with the host system in order for the customer to benefit from the advertised advantages of lower development and maintenance costs. Interestingly, when the claims of COTS advantages are closely examined, one

finds that to a great extent these COTS components consist of hardware and office products, not mission critical software [CLE97].

Obviously, COTS components are different from host components with respect to one or more of the following attributes: source, development paradigm, safety, reliability, maintainability, availability, security, and other attributes. However, the important question is whether they should be treated differently when deciding to deploy them for operational use; we suggest the answer is *no*. We use *reliability* as an example to justify our answer. In order to demonstrate its reliability, a COTS component must pass the same reliability evaluations as the host components, otherwise the COTS components will be the weakest link in the chain of components and will be the determinant of software system reliability. The challenge is that there will be less information available for evaluating COTS components than for host components but this does not mean we should despair and do nothing. Actually, there is a lot we can do even in the absence of documentation on COTS components because the customer will have information about how COTS components are to be used in the host system. To illustrate our approach, we will consider the reliability, maintainability, and availability (RMA) of COTS components as used in host systems.

In addition, COTS suppliers should consider increasing visibility into their products to assist customers in determining the components' fitness for use in a particular application. We offer ideas about information that would be useful to customers and what vendors might do to provide it.

This paper is organized as follows: reliability, maintainability, availability, requirements risk analysis, improved visibility into COTS, cost as the universal COTS metric, and conclusions.

Reliability

There are some intriguing questions concerning how to evaluate the reliability of COTS components that we will attempt to answer [SCH991]. Among these are the following: How do we estimate the reliability of COTS when there is no data available from the vendor? How do we estimate the reliability of COTS when it is embedded in a host system? How do we revise our reliability estimates once COTS has been

upgraded? A fundamental problem arises in assessing the reliability of a software component: a software component will exhibit different reliability performance in different applications and environments. A COTS component may have a favorable reliability rating when operated in isolation but a poor one when integrated in a host system. What is needed is the operational profile of COTS components as integrated into the host system in order to provide some clues as to how to test COTS components. We will assume the worst-case situation that documentation and source code are not available. Thus, inspection would not be feasible and we would have to rely exclusively on testing and reliability calculations derived from test data to assess reliability.

The operational profile identifies the criticality of components and their duration and frequency of use. Establishing the operational profile leads to a strategy of what to test, with what intensity, and for what duration. We must recognize that a COTS component must be tested with respect to *both* its operational profile and the operational profile of the host system of which it is a part. The COTS component would be treated like a black box for testing purposes similar to a host component being delivered by design to testing but without the documentation. Testing the COTS components according to these operational profiles will produce failure data that can be used for two purposes: 1) make an empirical reliability assessment of COTS components in the environment of the host system and 2) provide data for estimating the parameters of a reliability model for predicting future reliability [SCH97].

A comprehensive software reliability engineering process is described in [ANS93]. As pointed out by Voas, black box and operational testing alone may be inadequate [VOA98]. In addition, he advocates using fault injection to corrupt one component (e.g., COTS component) to see how well other components (e.g., the host system) can tolerate the failed component. While this approach can identify problems in the software, it cannot fix them without documentation. Thus there must be a contract with the vendor that allows the customer to report problems to the vendor for their resolution. Unfortunately, from the customer's standpoint, vendors are unlikely to agree to such an arrangement unless the customer has significant leverage such as the Federal Government. In the

271

case where documentation is available, it would be subjected to a formal inspection of its understandability and usability. If the documentation satisfies these criteria, it would be used as an aid to inspecting any source code that might be available. Next we consider COTS maintainability issues.

Maintainability

In the case of maintainability, there are more intriguing issues. Suppose a problem occurs in a host system. Is the problem in COTS or in the host software? Suppose it is caused by an interaction of the two. The customer knows the problem has occurred, but does not know how to fix it if there is no documentation. The vendor, not being on site, does not know the problem has occurred. Even the vendor may not know how to fix the problem if the source of the problem is the host software or an interaction between it and COTS components. In addition, suppose the customer needs to upgrade the host software and this upgrade is incompatible with the COTS components. Or, conversely, the vendor upgrades COTS components and they are no longer compatible with the host software. Lastly, suppose there are no incompatibilities, but the customer may be forced to install the latest COTS components upgrade in order to continue to receive support from the vendor. None of these situations can be resolved without either the customer having documentation to aid in fixing the problem, or a contract with the vendor of the type mentioned above. As in the case of reliability, when neither of these remedies is available, problems can only be identified but they cannot be fixed. Thus the software cannot be maintained. An additional factor that impacts both reliability and maintainability is that the vendor is unlikely to continue to support the software if the customer modifies it. Thus the situation degenerates to one in which the customer is totally dependent on vendor support to achieve reliability and maintainability objectives. This may be satisfactory for office product applications but it is unsatisfactory for mission critical applications. Next we consider the COTS availability issues.

Availability

High availability is crucial to the success of a mission critical system. What will be system availability using COTS? To attempt to answer this question, it is useful to consider hardware as a frame of reference. The ultimate COTS is hardware; it has interchangeable and replacement components. Maintenance costs are kept low and availability is kept high by replacing failed components with identical components. Unlike hardware, availability cannot be kept high by "replacing" the software. A failed component cannot be replaced because the replacement component would have the same fault as the failed component. Fault tolerant software is a possibility but it has had limited success. We see that availability is a function of reliability and maintainability as related by the formula:

Availability = $MTTF/(MTTF+MTTR)$ =

$1/1+(MTTR/MTTF)$,

where MTTF is mean time to failure and MTTR is mean time to repair. MTTF is related to reliability and MTTR is related to maintainability. For high availability, we want to drive *time to failure* to infinity and *repair time* to zero. However, we have seen from the discussion of reliability and maintainability that achieving these objectives is problematic. Thus to achieve high availability, either the COTS software must be of high intrinsic reliability – probably a naive assumption – or there must be in place a strong vendor maintenance program (this assumption may be equally naive). Next we consider COTS visibility issues.

Improved Visibility into COTS

Major drawbacks of including COTS in a software system are the lack of visibility into how the COTS components were developed and an incomplete understanding of the components' behavioral properties [SCH991]. Without this information, it is difficult to assess COTS components to determine their fitness for a particular application. As suggested by McDermid in [TAL98], a partial solution might be for COTS vendors to identify a set of behavioral properties that should be satisfied by the software, and then certifying that those properties are satisfied. For instance, an operating system supplier might certify that a lower-priority task does not interrupt a higher priority task as long as the higher priority task holds the resources required to continue processing. COTS vendors might also include the specifications of those components as well as

details of verification activities in which those specifications had been used to show that specific behavioral properties of the software were satisfied. For instance, an effort in progress at the Jet Propulsion Laboratory [JPL98] involves developing libraries of reusable specifications for spacecraft software components using the PVS specification language [SRI98]. The developers of the libraries work cooperatively with anticipated customers to develop the specifications and identify those properties that the components should satisfy. As they develop the libraries, the component developers use the PVS theorem proverb to show that the behavioral properties are satisfied by the specification. These proofs are intended to be distributed with the libraries. When customers modify the libraries, perhaps to customize them for a new mission, they will be able to use the accompanying proofs as a basis for showing that the modified specification exhibits the desired behavioral properties. Similarly, commercial vendors could work with existing and potential customers through user groups to discover those behavioral properties in which users are the most interested, and then work to certify that their components satisfy those properties. Next we present a methodology for analyzing requirements risk when COTS is embedded in a host system.

Requirements Risk Analysis

In this section we first describe the Shuttle risk management process. Then we consider how it could be modified to accommodate the use of COTS. In providing this analysis, it should not be inferred that we necessarily advocate the use of COTS on the Shuttle or on any other safety critical system. Whether COTS should be employed would depend upon many environmental and application factors. Rather, our goal is to investigate whether the Shuttle risk analysis process is adaptable to the use of COTS.

Shuttle Risk Management Process

One of the software development and maintenance problems of the NASA Space Shuttle Flight Software organization is to evaluate the risk of implementing requirements changes. These changes can affect the reliability, availability and maintainability of the software. To assess the risk of change, a number of risk factors are used. The risk factors were identified by agreement between

NASA and the development contractor based on assumptions about the risk involved in making changes to the software. This formal process is called a risk assessment. No requirements change is approved by the change control board without an accompanying risk assessment. During risk assessment, the development contractor will attempt to answer such questions as: "Is this change highly complex relative to other software changes that have been made on the Shuttle?" If this were the case, a high-risk value would be assigned for the complexity criterion. To date this qualitative risk assessment has proven useful for identifying possible risky requirements changes or, conversely, providing assurance that there are no unacceptable risks in making a change.

The following are the definitions of the risk factors, where we have placed the factors into categories and have provided our interpretation of the question the factor is designed to answer. In addition, we added the risk factor *requirements specifications techniques* because we feel that this one could represent the highest reliability risk of all the factors if a technique leads to misunderstanding of the intent of the requirements. For each of the risk factors, we analyze its appropriateness for COTS. As you will see, this analysis not only determines the adaptability of the process to COTS, but also exposes some serious issues in the employment of COTS in *any* system. For example, the Shuttle risk process is all about assessing the risk of requirements changes. In COTS, we would not want to attempt changes because we don't have the necessary source code and other documentation. Furthermore, if we did make a change, it could invalidate our software license. This situation illuminates a serious deficiency in using COTS. Therefore, our only recourse, if feasible, is to change the host software to reflect the change. In other words, COTS has to be used "*as is*" in our system. Thus, in what follows, the risk factors are a function of the change in the host software and how the change relates to and can be integrated with COTS.

In order to modify the Shuttle risk process to make it applicable to the use of COTS, we must change the software change metric from lines of code to components. In addition, we must change our view of the software from a set of individual instructions to a set of interconnected components. Otherwise, it would make no sense

to talk about number of lines of code to be changed in the host software when we only have visibility of COTS at the component level. We will also assume an object oriented development and maintenance paradigm.

Requirements Change Risk Factors

The following are the definitions of the Shuttle risk factors modified to accommodate the use of COTS, where, as mentioned previously, only *host software components* can be changed, but in making the changes, the relationship with COTS components must be considered. If the answer to a yes/no question is "yes". it means this is a high-risk change with respect to the given factor. If the answer to a question that requires an estimate is an anomalous value, it means this is a high-risk change with respect to the given factor. When a change to a component is mentioned below, it will be understood to be a change to host software.

Complexity Factors

o   Qualitative assessment of complexity of change (e.g., very complex)

- Is this change highly complex relative to other software changes that have been made on the system? What are the interfaces between the host components and COTS components that are affected by the change? Is the change more complex for the host system than for the host software alone?

o Number of modifications or iterations on the proposed change

- How many times must the change be modified or presented to the Change Control Board (CCB) before it is approved?

Size Factors

o   Number and types of components affected by the change

- How many components and types of components must be changed to implement the requirements change?

o   Size of software components that are affected by the change

- How many component objects are affected by the change?

Criticality of Change Factors

o Whether the software change is on a nominal or off-nominal component path (i.e., exception condition)

- Will a change to an off-nominal component path affect the reliability of the software?

o   Operational phases affected by the changed component path (e.g., ascent, orbit, and landing)

- Will a change to a critical phase of the mission (e.g., ascent and landing) affect the reliability of the software?

Locality of Change Factors

o   The area of the affected change (i.e., critical area such as a component path for a mission abort sequence)

- Will the change affect objects of components that are critical to mission success?

o   Recent changes to components in the area affected by the requirements change

- Will successive changes to the components in a given area lead to non-maintainable code?

o   New or existing components that are affected

- Will a change to new components (i.e., a change on top of a change) lead to non-maintainable software?

o   Number of system or hardware failures that would have to occur before the components that implement the requirement are executed

- Will the change be on a component path where only a small number of system or hardware failures would have to occur before the changed components are executed ?

Requirements Issues and Function Factors

o Number and types of other requirements affected by the given requirement change (requirements issues)

- Are there other requirements that are going to be affected by this change? If so, these requirements will have to be resolved before implementing the given requirement.

o Possible conflicts among requirements changes (requirements issues)

- Will this change conflict with other requirements changes (e.g., lead to conflicting operational scenarios)

o Number of principal software functions and components affected by the change

- How many major software functions and components will have to be changed to make the given change?

Performance Factors

o Amount of memory required to implement the change

- Will the change use memory to the extent that other functions and components will not have sufficient memory to operate effectively?

o Effect on CPU performance

- Will the change use CPU cycles to the extent that other functions and components will not have sufficient CPU capacity to operate effectively?

Personnel Resources Factors

o Number of inspections of components and objects required to approve the change

- Will the number and duration of inspections be significant?

o Manpower required to implement the change

- Will the manpower required to implement the software change be significant?

o Manpower required to verify and validate the correctness of the change

- Will the manpower required to verify and validate the software change be significant?

Tools Factor

o Software tools creation or modification required to implement the change

- Will the implementation of the change require the development and testing of new tools – for example the development of component and object testing tools?

o Requirements specifications techniques (e.g., flow diagram, state chart, pseudo code, control diagram).

- Will the requirements specification method be difficult to understand and translate into components and objects?

As an example, Table 1 shows a partial list of the risk factors compiled for the for the Shuttle *Three Engine Out Auto Contingency* and *Single Global Positioning System* requirements changes.

Table 1

Change Request Number	SLOC Changed	Complexity Rating of Change	Criticality of Change	Number of Principal Functions Affected	Number of Modifications Of Change Request	Number of Requirements Issues	Number of Inspections Required	Manpower Required to Make Change
107734	1933	4	3	27	. 7	238	12	209.3 MW

Discussion

Although we believe we have made a reasonable translation from a code oriented requirements risk analysis to a component oriented one, it is not clear that the resultant risk model would be entirely usable because no matter how we define the software entities of interest, we still do not have equal visibility of the host

software and COTS. We suggest this is a fundamental problem that has not been solved by COTS advocates, particularly for safety critical systems. Next we present a framework for identifying and analyzing the cost of COTS.

Cost as the Universal COTS Metric

We focus on factors that the user should consider when deciding whether to use COTS software [SCH992]. We take the approach of using the common denominator *cost*. This is done for two reasons: first, cost is obviously of interest in making such decisions and second a single metric − cost in dollars − can be used for evaluating the pros and cons of using COTS. The reason is that various software system attributes, like acquisition cost and availability (i.e., the percentage of scheduled operating time that the system is available for use), are non-commensurate quantities. That is, we cannot relate quantitatively "a low acquisition cost" with "high availability". These units are neither additive nor multiplicative. However, if it were possible to translate availability into either a cost gain or loss for COTS software, we could operate on these metrics mathematically. Naturally, in addition to cost, the user application is key in making the decision. Thus one could develop a matrix where one dimension is *application* and the other dimension is the various *cost elements*. We show how cost elements can be identified and how cost comparisons can be made over the *life* of the software. Obviously, identifying the costs would not be easy. The user would have to do a lot of work to set up the decision matrix but once it was constructed, it would be a significant tool in the evaluation of COTS. Furthermore, even if all the required data cannot be collected, having a framework that defines software system attributes would serve as a user guide for factors to consider when making the decision about whether to use COTS software or in-house developed software. Note that host software could be developed either in-house or under contract. If the former, the in-house cost element below apply to host software.

Certainly, different applications would have varying degrees of relationships with the cost elements. For example, flight control software would have a stronger relationship with the cost of unavailability than a spreadsheet application. Conversely, the latter would have a stronger relationship with the cost of inadequacy of tool features than the former. Due to the difficulty of identifying specific COTS-related costs, our initial approach is to identify cost elements on the ordinal scale. Thus, the first version of the decision matrix would involve ordinal scale metrics (i.e., the cost of unreliability is more important for flight control software than for spreadsheet applications). As the field of COTS analysis matures and as additional data is collected about the cost of using COTS, we will be able to refine our metrics to the ratio scale (e.g., the cost of unreliability in a host system is two times that in a commercial COTS system).

The cost elements for comparing COTS software with in-house software are identified below. This list is not exhaustive; its purpose is to illustrate the approach. These elements apply whether we are comparing a system comprised of all COTS components with all in-house components or comparing only a subset of COTS components with corresponding in-house components. Explanatory comments are made where necessary. Mean values are used for some quantities in the initial framework. This is the case because it will be a challenge to collect *any* data for some applications. Therefore, the initial framework should not be overly complex. Variance and statistical distribution information could be included as enhancements if the initial framework proves successful.

Cost Elements

$C_c(j)$ = Cost of acquiring COTS software in year j.

$C_i(j)$ = Cost of developing in-house software in year j.

$U_c(j)$ = Cost of upgrading COTS software in year j.

$U_i(j)$ = Cost of upgrading in-house software in year j.

$P(j)$ = Cost of personnel who use the software system in year j. This quantity represents the value to the customer of using the software system.

$M_c(j)$ = Cost per unit time of repairing a fault in COTS software in year j. This is the cost of customer time involved in resolving a problem with the vendor.

$M_i(j)$ = Cost per unit time of repairing a fault in in-house software in year j.

$R_c(j)$ = Mean time of repairing a fault that causes a failure in COTS software in year j. This is the average time that the user spends in resolving a problem with the vendor.

$R_i(j)$ = Mean time of repairing a fault that causes a failure in in-house software in year j.

$T(j)$ = Scheduled operating time for the software system in year j.

$A_c(j)$ = Availability of software system that uses COTS software in year j.

$A_i(j)$ = Availability of software system that uses software developed in-house in year j.

These quantities are the fractions of $T(j)$ that the software system is available for use.

$F_c(j)$ = Failure rate of COTS software in year j.

$F_i(j)$ = Failure rate of in-house software in year j.

These quantities are the number of failures per year that cause loss of productivity and availability of the software system.

In some applications, some or all of the above quantities may be known or assumed to be constant over the life of the software system. Using the above cost elements, we derive the equations for the annual costs of the two systems and the difference in these costs. In the cost difference calculations that follow, a positive quantity is favorable to in-house development and a negative quantity is favorable to COTS.

Cost of Acquiring Software

Difference in annual cost = $C_c(j) - C_i(j)$ (1)

Cost of Upgrading Software

Difference in annual cost = $U_c(j) - U_i(j)$ (2)

Cost of Software being Unavailable for Use

Annual cost of COTS software being unavailable for use = $(1 - A_c(j)) * P(j)$.

Annual cost of the in-house software being unavailable for use = $(1 - A_i(j)) * P(j)$.

Difference in annual cost =
$P(j) * (A_i(j) - A_c(j))$ (3)

Cost of Repairing Software

Average annual cost of repairing failed COTS software = $F_c(j) * T(j) * R_c(j) * M_c(j)$.

Average annual cost of repairing failed in-house software = $F_i(j) * T(j) * R_i(j) * M_i(j)$.

Difference in annual cost =

$T(j) * ((F_c(j) * R_c(j) * M_c(j)) - ((F_i(j) * R_i(j) * M_i(j))$ (4)

Then, TC_j, total difference in cost in year j, is the sum of (1), (2), (3), and (4). Because there is the opportunity to invest funds in alternate projects, costs in different years are not equivalent (i.e., funds available today have more value than an equal amount in the future because they could be invested today and earn a future return). Therefore, a stream of costs over the life of the software for n years must be discounted by k, the rate of return on alternate use of funds. Thus the total discounted cost differential between COTS software and in-house software is:

$$\sum_1^n TC_j / (1+k)^j$$

In this initial formulation, we have not included possible differences in functionality between the two approaches. However, a reasonable assumption is that COTS software would not be considered unless it could provide minimum functionality to satisfy user requirements. Thus, a typical decision for the user is whether it is worth the additional life cycle costs to develop an in-house software system with all the desirable attributes.

Conclusions

The decision to employ COTS on mission critical systems should not be based on development cost alone. Rather, costs should be evaluated on a total life cycle basis and RMA should be evaluated in a system context (i.e.,

COTS components embedded in a host system). COTS suppliers should also consider making available more detailed information regarding the behavior of their systems, and certifying that their components satisfy a specified set of behavioral properties. In addition, a formal risk assessment of requirements should be performed taking into account the characteristics of host system environments.

References

[ANS93] Recommended Practice for Software Reliability, R-013-1992, American National Standards Institute/American Institute of Aeronautics and Astronautics, 370 L'Enfant Promenade. SW, Washington, DC 20024, 1993.

[CLE97] Clemins, Archie, "IT-21: The Path to Information Superiority." CHIPS Jul 1997, http://www.chips.navy.mil/chips/archives/97_jul/file.htm, p. 1.

[JPL98] "Reusable Libraries of Formal Specifications", NASA Formal Methods web site, http://eis.jpl.nasa.gov/quality/Formal_Methods/library.html, 1998.

[KOH99] Ronald J. Kohl, "V&V of COTS Dormant Code: Challenges and Issues", Proceedings of the First Workshop on Ensuring Successful COTS Development, 21st International Conference on Software Engineering, Los Angeles, California, May 22nd, 1999. 2 pages.

[SCH97] Norman F. Schneidewind, "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, Vol. 46, No.1, March 1997, pp.88-98.

[SCH991] Norman F. Schneidewind and Allen P. Nikora, "Issues and Methods for Assessing COTS Reliability, Maintainability, and Availability", Proceedings of the First Workshop on Ensuring Successful COTS Development, 21st International Conference on Software Engineering, Los Angeles, California, May 22nd, 1999, 4 pages.

[SCH992] Norman F. Schneidewind, "Cost Framework for COTS Evaluation", Proceedings of COMPSAC 99, Phoenix, AZ, 27 October 1999, pp. 100-101.

[SRI98] "The PVS Specification and Verification System", SRI International Computer Science Laboratory, http://www.csl.sri.com/sri-csl-pvs.html, 1998.

[TAL98] Nancy Talbert, "The Cost of COTS", IEEE Computer, Vol. 31, No. 6, June 1998, pp. 46-52.

[VOA98] Jeffrey M. Voas, "Certifying Off-the-Shelf Software Components", IEEE Computer, Vol. 31, No. 6, June 1998, pp. 53-59.

Knowledge-Based
SYSTEMS

Approximate declarative semantics for rule base anomalies

Du Zhang[a,*], Luqi[b]

[a]Department of Computer Science, California State University, Sacramento, CA 95819-6021, USA
[b]Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA

Abstract

Despite the fact that there has been a surge of publications in verification and validation of knowledge-based systems and expert systems in the past decade, there are still gaps in the study of verification and validation (V&V) of expert systems, not the least of which is the lack of appropriate semantics for expert system programming languages. Without a semantics, it is hard to formally define and analyze knowledge base anomalies such as inconsistency and redundancy, and it is hard to assess the effectiveness of V&V tools, methods and techniques that have been developed or proposed. In this paper, we develop an approximate declarative semantics for rule-based knowledge bases and provide a formal definition and analysis of knowledge base inconsistency, redundancy, circularity and incompleteness in terms of theories in the first order predicate logic. In the paper, we offer classifications of commonly found cases of inconsistency, redundancy, circularity and incompleteness. Finally, general guidelines on how to remedy knowledge base anomalies are given. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Knowledge base anomalies; Inconsistency; Redundancy; Circularity; Incompleteness; Knowledge base verification

1. Introduction

The last decade has witnessed a surge of publications in verification and validation (V&V) of expert systems and knowledge-based systems which resulted in several books [1,2], and special issues of several journals [3–6]. Major AI conferences have had workshops and special sessions that were devoted to the issue. A sample of additional publications can be found in Refs. [7–40]. Many V&V methods, techniques and tools have been proposed, developed or implemented for expert system applications. On the other hand, advances in knowledge engineering have resulted in better methodologies and practice that aim at reducing errors and faults during system development and maintenance [41–44]. Despite all these activities, there are still gaps in the study of V&V of expert systems, not the least of which is the lack of appropriate semantics for expert system programming languages. Without a semantics, it is hard to formally define and analyze knowledge base (KB) anomalies such as inconsistency and redundancy, and it is hard to assess the effectiveness of V&V tools, methods and techniques that have been developed or proposed.

V&V of expert systems in general and V&V of KB in particular need to be based on a sound theoretical foundation. However, the reality is that "the construction of either declarative or Hoare-style semantics for current rule-based languages is a hopeless task" [31]. In the long run, concern for verifiability and reliability should lead to the development of programming languages with tractable semantics for expert system applications. In the meantime, some approximate semantics (declarative or imperative) is needed to enable a formal analysis of properties of expert system components (such as a KB). For example, sketches of an approximate declarative semantics, which is based on a logical interpretation of a rule base, and an approximate imperative semantics, which is based on axiomatic logic and invariants, for the current rule-based programming languages were proposed in Ref. [31].

Adopting a declarative semantics for a rule-based language has some potential difficulties: (a) It is hard to provide a purely declarative interpretation of rules, because they often behave in an imperative manner with the intended side effects of updating a working memory. Simply treating a rule base as a logical theory may result in an excessively conservative semantics. (b) Due to the fact that consistency in the first order logic is semi-decidable, there does not exist an algorithm that can find all inconsistencies and redundancies in an arbitrary first order KB, thus, making it difficult to develop practical V&V tools.

* Corresponding author. Tel.: − 1-916-278-7952; fax: + 1-916-278-6774.

E-mail addresses: zhangd@ecs.csus.edu (D. Zhang), luqi@cs.nps.navy.mil (Luqi)

Table 1
Typesetting conventions

Symbol	Meaning
Đ	A nonempty domain of elements
ζ	An interpretation
Boldface capital letter	Set of wff (literals), or set of rules
Ordinary capital letter	Individual wff (literal)
Lower-case ordinary letter	Constant
Lower-case italic letter(s)	Predicate
r	Rule label
f	Fact label
LHS (r_i)	Set of literals in the left-hand side of r_i
RHS (r_i)	Set of literals in the right-hand side of r_i
true, false	Logical values
x, y, z, x', y', z'	Variable

There have been several efforts toward providing a precise characterization of the logical nature of a rule-based KB [11,31,35]. An algorithm to detect all inconsistencies and redundancies in "a certain well-defined, reasonably expressive, subset of all quasi-first-order-logic KB" is presented in [11].[1] The results in [35] indicate that a rule-based language is still amenable to logical analysis.

The purposes of this paper are to (a) Provide an approximate declarative semantics for rule-based KB so that various KB anomalies can be formally defined and correctly understood. We go beyond the results of [11,31,35] by dealing with not only KB inconsistencies and redundancies, but also KB circularity and incompleteness. (b) Establish KB anomaly analysis procedures using theories in the first order predicate logic (such as the *model theory, satisfiability,* and *derivability* of certain tautologous well-formed formulas [45–47]). This may serve as the theoretical underpinnings of practical V&V tools. (c) Offer classifications for cases of inconsistency, redundancy, circularity and incompleteness commonly found in rule-based KB. (d) Propose guidelines on how to remedy the anomalies once they are identified.

The rest of the paper is organized as follows: Section 2 briefly reviews the terms and concepts to be used throughout the paper. Definitions, classifications and analyses of KB inconsistency, redundancy, circularity and incompleteness are provided in Sections 3–6, respectively. Some possible remedial measures for KB anomalies are discussed in Section 7. Section 8 concludes with remarks about future work.

---

[1] The key step in the algorithm is the subsumption tests which must be decidable for a given KB in order for the KB to be completely analyzed for inconsistency and redundancy. The subsumption tests will be decidable only when the expressions to be tested satisfy the *quantifier decoupled* (q-decoupled) property [11]. In general, one does not know in advance if a given KB will generate any non q-decoupled expressions because there does not exist a syntactic test for determining the q-decoupleability of the KB.

2. Preliminaries

We assume that the reader is familiar with the basic concepts and terminology in the first order predicate logic [45–47]. We use *wff* to denote the *well-formed formulas* of the predicate logic. An *atomic formula* (or *atom*) refers to an *n*-place predicate symbol and its *n* terms. A *ground* atom is one not containing any variables. A *literal* is an atom or its negation. To avoid confusion, we adopt the typesetting conventions as given in Table 1.

Definition 1. An *interpretation* of a wff consists of a non-empty domain Đ, and an assignment of "values" to each constant, function symbol and predicate symbol appearing in the wff according to the following: (a) assigning an element of Đ to each constant; (b) assigning a mapping from $Đ^n$ to Đ to each *n*-ary function symbol; and (c) assigning a mapping from $Đ^n$ to {*true, false*} to each *n*-ary predicate symbol.

Definition 2. A wff H (or a set Ç of wff) is *satisfiable (consistent)* if and only if there exists an interpretation such that H (or every wff in Ç) is evaluated to *true* for all variable assignments[2] under ζ, which is denoted $\models_\zeta$ H (Ç). ζ is said to be a *model* of H (Ç) and ζ *satisfies* H (Ç). H (Ç) is *inconsistent* if and only if there exists no model for H (Ç). H is said to be *valid (tautologous)* if and only if every possible interpretation satisfies H. H is a *logical consequence* of Ç if and only if every model of Ç is also a model of H. This is denoted as Ç ⊨ H.

Theorem 1. Given a set of wff Ç = {P, …, Q} and a wff H, Ç ⊨ H if and only if P ∧ … ∧ Q → H is valid.

Definition 3. Let Ç and Ç′ be sets of wff. Ç ≈ Ç′ denotes that Ç is satisfiable if and only if Ç′ is satisfiable [45].

This paper focuses on rule-based knowledge bases. A rule-based KB can be divided into a set of *facts* which are stored in a *working memory* (WM) and a set of *rules* stored in a *rule base* (RB). Rules represent general knowledge about an application domain. They are entered into a KB during initial knowledge acquisition or subsequent updates. Facts in a WM provide specific information about the problems at hand and may be elicited either dynamically from the user during each problem-solving session or statically from the domain expert during knowledge acquisition process, or derived through rule deduction.

---

[2] A variable assignment is a mapping from variables in a wff to elements in Đ.

Table 2
Same, synonymous, complementary, mutual exclusive, incompatible, and conflict literals

Semantics	Syntax	
	Identical	Different
Equivalent	*Same:* denoted as $L_1 = L_2$[a]. L_1 and L_2 are syntactically identical (same predicate symbol, same arity, and same terms at corresponding positions)	*Synonymous:* denoted $L_1 \cong L_2$[b]. L_1 and L_2 are syntactically different, but logically equivalent
Conflict[c]	*Complementary:* denoted $L_1 \# L_2$. L_1 and L_2 are an atom and its negation	*Mutual exclusive:* denoted $L_1 \ominus L_2$. L_1 and L_2 are syntactically different and semantically have opposite truth values
		Incompatible: denoted $L_1 \asymp L_2$. L_1 and L_2 are complementary pair of synonymous literals

[a] Given two rules r_i and r_j, if $LHS(r_i) = \{P1, ..., Pn\}$ and $LHS(r_j) = \{P1', ..., Pn'\}$, then $LHS(r_i) = LHS(r_j)$ iff $\forall i \in [1,n] Pi = Pi'$.
[b] Given two rules r_i and r_j, if $LHS(r_i) = \{P1, ..., Pn\}$ and $LHS(r_j) = \{P1', ..., Pn'\}$, then $LHS(r_i) \cong LHS(r_j)$ iff $\forall i \in [1,n] Pi \cong Pi'$.
[c] L_1 and L_2 are conflict literals, denoted $L_1 \overline{\natural} L_2$, if $(L_1 \# L_2) \vee (L_1 \ominus L_2) \vee (L_1 \asymp L_2)$.

Definition 4. Rules in a KB have the format: $P_1 \wedge ... \wedge P_n \rightarrow R$, where P_i's are the conditions (collectively, the *left-hand side*, LHS, of a rule). R is the conclusion (or *right-hand side*, RHS, of a rule), and the symbol " $\rightarrow$ " is understood as the logical implication. The P_i's and R are *literals*. If the conditions of a rule instance are satisfied by facts in WM, then its conclusion is deposited into WM.

Definition 5. A fact is represented as a ground atom. It specifies an instance of a relationship among particular objects in the problem domain. WM contains a collection of *positive* ground atoms, which are deposited through either assertion (initial or dynamic), or rule deduction.

Definition 6. A negated condition $\neg p(x)$ in the LHS of a rule is satisfied if $p(x)$ is not in WM for any x. A negated ground atom $\neg p(a)$ in the LHS of a rule is satisfied if $p(a)$ is not in WM. A negated conclusion $\neg R$ in the RHS of a rule results in the removal of R from WM, when the LHS of the rule is satisfied.[3] Rule instances and negated literals can be utilized by the inference system, but are never deposited into WM [11].

Definition 7. Given two sets of literals L and L'. L' is said to be a *specialization* of L. denoted $L' \trianglelefteq L$, if there exists a nonempty set of *substitutions* θ, such that $L' = (L)\theta$. In particular, a literal P' is a specialization of P. denoted as $P' \leq P$ if there exists a nonempty set of substitution θ such that $P' = (P)\theta$.

Definition 8. Given a set L of n literals, $\rho(L)$ represents the set of all literal permutations in L.

Definition 9. If r_i is a rule and P is a literal, the expression $r_i \vdash P$ is used to indicate arbitrary length derivation of P from r_i in terms of some inference methods.[4]

Using *logical equivalence*, we can always convert a logical implication into a disjunction of literals. We further simplify the notation by dropping the logical connective " $\vee$ " from such a disjunction. For instance, the set of wff $\{P \wedge Q \rightarrow R, U \wedge \neg V \rightarrow W\}$ has the following logically equivalent short representation: $\{\neg P \neg QR, \neg UVW\}$ where each element in the set is a disjunction of literals.

Definition 10. The concepts of the *same, synonymous, complementary, mutual exclusive, incompatible,* and *conflict* literals are defined in Table 2 in terms of syntax and semantics considerations.

Example 1. Given the following literals: *father*(x, john), *male_parent*(x, john), *animal*(sea_cucumber), *vegetable*(sea_cucumber), *bird*(fred), $\neg bird$(fred), *sent_to*(x, emergency_room), *sent_to*(x, waiting_room), *expensive*(x), *high_priced*(x), we have:

father(x, john) = *father*(x, john);
father(x, john) $\cong$ *male_parent*(x, john);
bird(fred) $\#$ $\neg bird$(fred);
animal(sea_cucumber) $\ominus$ *vegetable*(sea_cucumber);
sent_to(x, emergency_room) $\ominus$ *sent_to*(x, waiting_room);
expensive(x) $\asymp$ $\neg high\_priced$(x);
father(x, john) $\overline{\natural}$ $\neg male\_parent$(x, john).

---

[3] There would be no effect on WM if R is not in WM when $\neg R$ is derived.

[4] Strictly speaking, the expression should be $\{r_i \cup WM\} \vdash P$ because facts in WM will be used during the derivation.

281

Table 3
Types of inconsistency

Type	Description	Pattern
I-1	Rules with the same LHS result in complementary conclusions	$LHS(r_i) = LHS(r_j)$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \neq Q$
I-2	Rules with shared condition(s) result in complementary conclusions	$LHS(r_i) \cap LHS(r_j) = C$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \neq Q$
I-3	Rules with the same LHS result in mutual exclusive conclusions	$LHS(r_i) = LHS(r_j)$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \doteq Q$
I-4	Rules with shared condition(s) result in mutual exclusive conclusions	$LHS(r_i) \cap LHS(r_j) = C$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \ominus Q$
I-5	Rules with the same LHS result in incompatible conclusions	$LHS(r_i) = LHS(r_j)$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \bowtie Q$
I-6	Rules with shared condition(s) result in incompatible conclusions	$LHS(r_i) \cap LHS(r_j) = C$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \bowtie Q$
I-7	Rules with synonymous LHS result in complementary conclusions	$LHS(r_i) \cong LHS(r_j)$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \neq Q$
I-8	Rules with shared synonymous conditions result in complementary conclusions	$L \subseteq LHS(r_i)$ and $L' \subseteq LHS(r_j)$ and $L \cong L'$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \neq Q$
I-9	Rules with synonymous LHS result in mutual exclusive conclusions	$LHS(r_i) \cong LHS(r_j)$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \doteq Q$
I-10	Rules with shared synonymous conditions result in mutual exclusive conclusions	$L \subseteq LHS(r_i)$ and $L \subseteq LHS(r_j)$ and $L \cong L'$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \doteq Q$
I-11	Rules with synonymous LHS result in incompatible conclusions	$LHS(r_i) \cong LHS(r_j)$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \bowtie Q$
I-12	Rules with shared synonymous conditions result in incompatible conclusions	$L \subseteq LHS(r_i)$ and $L \subseteq LHS(r_j)$ and $L \cong L$ and $r_i \vdash P$ and $r_j \vdash Q$, where $P \bowtie Q$
I-13	Rules with consistent LHS result in complementary conclusions	$= \{LHS(r_i), LHS(r_j)\} \wedge LHS(r_i) \cap LHS(r_j) = C \wedge r_i \vdash P$ and $r_j \vdash Q$, where $P \neq Q$
I-14	Rules with consistent LHS result in mutual exclusive conclusions	$= \{LHS(r_i), LHS(r_j)\} \wedge LHS(r_i) \cap LHS(r_j) = C \wedge r_i \vdash P$ and $r_j \vdash Q$, where $P \doteq Q$
I-15	Rules with consistent LHS result in incompatible conclusions	$= \{LHS(r_i), LHS(r_j)\} \wedge LHS(r_i) \cap LHS(r_j) = C \wedge r_i \vdash P$ and $r_j \vdash Q$, where $P \bowtie Q$
II-1	Rules with a condition result in complementary literal	$r \vdash Q$, where $P \in LHS(r) \wedge P \neq Q$
II-2	Rules with a certain condition result in incompatible literal	$r \vdash Q$, where $P \in LHS(r) \wedge P \bowtie Q$
II-3	Rules with a condition P result in mutual exclusive literal	$r \vdash Q$, where $P \in LHS(r) \wedge P \doteq Q$

In this paper. we do not consider the situation in which rules are augmented with *certainty factors*. Because of the way they are defined. rules and facts are subsets of wff. Therefore, the terms "rule" and "fact" can be freely replaced by the term "wff" throughout the rest of the paper.

3. KB inconsistency

3.1. Definition of inconsistency

The root cause of KB inconsistency is due to rules in RB.

but its manifestation is through WM. For instance. the inconsistency of a RB containing a pair of rules $\{p(x) \rightarrow q(x). p(x) \rightarrow \neg q(x)\}$ is not apparent until a fact $p(a)$ is asserted into WM. In general. although the rules in a RB may be consistent on their own (because there exists a model for them). they can form an inconsistent theory when combined with certain facts in WM. In order for a KB to be consistent. there needs to be a model for both RB and WM.

On the other hand. facts in WM are changing over time due to dynamic assertions and retractions. If we use subscripts to denote states of WM at different times. RB may be consistent with WM_i. but inconsistent with WM where $i \neq j$. Thus. relying on a particular WM state in verifying the consistency of RB may not produce an accurate result.

Definition 11. Let WM_0 and $R(WM)$ denote the initial state for WM and the reachability set of all possible WM states from WM_0. respectively. Let WM denote all legitimate facts[5] for an application. $WM = \cup \{WM_i | WM_i \in R(WM_0)\}$.

Definition 12. Given two interpretations ζ and ζ_i. ζ is an extension of ζ_i. denoted as $\zeta_i \sqsubseteq \zeta$. if the domain and assignments in ζ are retained in ζ_i.

Definition 13. Let ζ_0 be a model for WM.[6] A KB is *inconsistent* if and only if $\neg \exists \zeta [\zeta_0 \sqsubseteq \zeta \wedge \models_\zeta RB]$.

During problem solving process. inconsistent rules in RB allow derivations of conflicting (complementary. mutual exclusive and incompatible) outcomes from the same. synonymous or consistent conditions. thus. seriously compromising the reliability and correctness of knowledge-based systems.

3.2. Classification of inconsistency

Two types of inconsistency are classified in Table 3. Each type consists of a set of patterns and each pattern encompasses different cases. Type I contains anomalous situations where rules with the same or synonymous conditions result in conflict (complementary. mutual exclusive and incompatible) conclusions. Type II captures the scenarios where a chain of deduction involves a condition and a conclusion (at two ends of the chain) which are either complementary. or mutual exclusive. or incompatible. It is very important to recognize the types of inconsistency for several reasons: (a)

---

[5] Facts that satisfy the validity constraints of the application domain.

[6] If there are validity constraints on facts in WM. then the models considered are restricted to those that satisfy the constraints.

so that effective detection algorithms can be developed: (b) the completeness of the V&V tools can be measured.

The exhaustive nature of the classification can be considered by enumerating all cases that result in an unsatisfiable RB (Definition 13). The clue is the derivation of conflict literals by a RB or a derived literal being in conflict with a fact in WM. Due to space limit, we will skip a formal proof.

3.3. Analysis

Given a RB and a WM containing a set of rules and a set of facts, respectively, we can show that the KB is consistent by trying to find a model for it. The way we try to find a model for the KB is through considering an arbitrary interpretation ζ. If ζ satisfies the KB (i.e. ζ satisfies RB and WM), then ζ is a model for it: otherwise, there is no model for the KB. If a model is found, then the KB is consistent: otherwise, it is inconsistent. We show the analysis through some examples.

Example 2. Given a KB consisting of a RB = $\{r_1, r_2, r_3, r_4, r_5\}$ and a WM = $\{f_1, f_2, f_3\}$ shown below

$r_1 : P \wedge Q \rightarrow A \qquad f_1 : P$

$r_2 : R \wedge Q \rightarrow B \qquad f_2 : Q$

$r_3 : A \wedge B \rightarrow W \qquad f_3 : R$

$r_4 : A \rightarrow D$

$r_5 : B \rightarrow \neg D$

we can show that there is no model for the KB, thus, it is inconsistent.

Proof. We convert the KB into the set below

$\Omega_1 = \{\neg P \neg QA, \neg R \neg QB, \neg A \neg BW,$

$\neg AD, \neg B \neg D, P, Q, R\}$

Let ζ be any interpretation for Ω_1.

- If ζ is a model for Ω_1, then $\models_\zeta P$, $\models_\zeta Q$ and $\models_\zeta R$;
- According to the first two elements in Ω_1, there must be $\models_\zeta A$ and $\models_\zeta B$;
- Since $\models_\zeta A$ and $\models_\zeta B$, there must be $\models_\zeta D$ and $\models_\zeta \neg D$ in order for $\neg AD$ and $\neg B \neg D$ to be *true*. But this is impossible. As a result, one of the rules of $\neg AD$ and $\neg B \neg D$ must be *false* under ζ.
- Since ζ cannot satisfy all rules in Ω_1, it is not a model for Ω_1. Because ζ is an arbitrary interpretation, there is no model for Ω_1. Thus, the given KB is inconsistent. $\square$

The inconsistency in Example 2 is of type I-13 because r_4 and r_5 have different but consistent LHS and result in conflicting conclusions D and $\neg D$. The proof procedure

can be automated using the *resolution principle* where the derivation of an empty clause amounts to the failure of finding a model (or the presence of inconsistency in the KB). In practice, we can use the structure of the derivation generated by the resolution principle to extract a set of inconsistent rules.

The above example demonstrates an inconsistency in the current state of a KB. There is, however, another scenario in which the proof procedure yields a model for a KB, but there exists the potential of inconsistency in a possible future state of the KB. Consider the situation where fact f_3 is a legitimate input but is not present in the WM at the time of checking, the proof procedure will find a model for (KB − f_3) and conclude that it is consistent. (This coincides with the intuitive explanation that the conflicting conclusion $\neg D$ is not deducible because the LHS of r_2 cannot be satisfied). However, inconsistency arises when fact f_3 is asserted into WM. This phenomenon confirms our early arguments that:

- The cause of inconsistency stems from rules, but facts will help expose the inconsistency. Thus the inconsistency checking should involve both RB and WM.
- KB consistency can be either temporary or persistent. For instance, KB − f_3 is temporarily consistent until f_3 is asserted. Such a transient consistency is not a reliable indicator. What is needed is an ultimate consistency that guarantees that a KB will be consistent for all possible states.
- The set of all legitimate facts in an application domain usually changes with time. Given a time period, it is important to identify the set of all legitimate facts during the period in order to conclude whether a KB will be persistently consistent during the period.

Operationally, when a pair of conflicting conclusions is derived, it amounts to a fact retraction in WM. In a rule-based programming language, there are two types of fact retraction: *explicit* one through a language construct such as retract and *implicit* one through derivation of a negated fact and negation as absence rule for WM. The implicit fact retraction would be an indicator for RB inconsistency, but it is not a necessary condition for RB inconsistency. The reason is that in general, a rule-based system may not have the *Church–Rosser* property, therefore the derived facts by RB for the same initial facts in WM may not be unique. For instance, when both r_2 and r_5 are enabled, depending on the conflict resolution strategy used by the control component of the system, r_2 and r_5 can be fired in different order. As a result, different sets of output (derived facts) will be produced.

If f_3 is not a legal input, then rule r_2 can never be enabled because of the unsatisfiability of its LHS. As a result, the rule will be picked up by the incompleteness checking and classified as an incomplete case.

The Church–Rosser property of a rule-based system refers to the fact that the order in which rules are fired does not affect the final values produced [31].

Example 3. Given a KB containing the following rules and facts

$r_1 : P \wedge Q \rightarrow R$ $f_1 : P$

$r_2 : R \rightarrow W$ $f_2 : Q$

$r_3 : W \rightarrow A$

$r_4 : A \rightarrow \neg P$

we can show that there is no model for the KB, thus, the KB is inconsistent.

Proof. We convert the KB into the set $\Omega_2 = \{ \neg P \neg QR, \neg RW, \neg WA, \neg A \neg P, P, Q \}$

Let ζ be any interpretation for Ω_2.

- If ζ is a model for Ω_2, then $\models_\zeta P$ and $\models_\zeta Q$;
- There must be $\models_\zeta \neg A$, $\models_\zeta \neg W$ and $\models_\zeta \neg R$, respectively, in order for $\neg A \neg P$, $\neg WA$, and $\neg RW$ to be *true* under ζ;
- However, there must be $\models_\zeta R$ according to the first element in Ω_2. R and $\neg R$ cannot be both *true* under ζ. As a result, one of the clauses of $\neg P \neg QR$ and $\neg RW$ must be *false* under ζ.
- Since ζ cannot satisfy all rules in Ω_2, it is not a model for Ω_2. Because ζ is an arbitrary interpretation, there is no model for Ω_2. Thus, the given KB is inconsistent. $\Box$

The inconsistency in Example 3 is of type II-1 because r_1 has a condition P and results in the derivation of $\neg P$. Type II inconsistency not only introduces the logical contradiction into the inference process, it also has other pragmatic ramifications:

- In Example 3, the inconsistency involves a pair of complementary literals. When r_4 is fired, it causes P to be removed from WM, thus either preventing those rules that rely on P as input from being enabled or deactivating those rules that are enabled as a result of P.
- A list of synonymous literals and a list of mutual exclusive literals must be declared and maintained as a KB is being built and modified. In addition to Definition 6, the following should be used to maintain the validity of WM:

  If $(P \leftrightharpoons Q) \wedge (Q \in WM)$, then $KB \vdash P$ would result in $(WM - \{Q\}) \cup \{P\}$.
  If $(P \neq Q) \wedge (Q \in WM)$, then $KB \vdash P$ would result in $(WM - \{Q\})$.

- Computationally, when $\neg P$ is a derived fact, the inference engine will check not only for the presence of P in WM, but also the presence of some literal synonymous to P.[9] Alternatively, before a derived fact P gets deposited into WM, the inference system also need to check for the presence of Q in WM that is mutually exclusive to P.

---

[9] Definition 6 now needs to be modified to reflect the impact of synonymous literals on the occurrence of $\neg P$ in LHS or RHS of a rule.

Though the use of synonymous and mutual exclusive literals may aid the expressive power of the language, their potential complications in system correctness should never be underestimated and their computational cost should not be ignored. Therefore, the use of those literals, especially synonymous literals, should be judicious.

4. KB redundancy

4.1. Definition of redundancy

Though redundancy may not cause logical problems (i.e. with no effect on the set of deducible literals), it may lead to following situations where potential problems may arise:

- During KB maintenance or evolution, if one of the redundant rules is modified and the others remain unchanged, then the updated KB will not correspond to the intended change, and inconsistencies can be introduced as well;
- For a KB where no certainty factors are utilized, redundant rules may be enabled under a given state, thus resulting in performance slow down because all the enabled redundant rules may be fired, even though the firings of those redundant rules will yield the same set of literals (conclusions);
- For a KB containing certainty factors, redundancy will become a serious problem, the reason being that each redundant rule may be fired, resulting in multiple countings of the same information, which, in turn, erroneously increases the level of confidence assigned to the derived literals (conclusions). This may ultimately impact the set of deducible literals.

If redundancy is introduced by design to speed up some classes of frequent deductions, then it is usually confined to a subset of the cases (e.g. types I-2, I-3, I-5 in Table 4). We can always isolate those "useful" redundant rules, and weed out redundancy from the KB where there is supposed to be none.

Definition 14. For a set S of rules, we define a function ψ which returns the number of distinct literals in S. If both L and $\neg L$ are in S, they will be counted as two different literals.

Definition 15. Given a set S of rules, if we can construct a set S' of rules such that $S \approx S'$ and

(a) either $S' = S - \Delta$, where $\Delta \neq \varnothing$ and $\Delta \subset S$;
(b) or $S' = \phi(S)$, where ϕ is a transformation on S such that $|S'| = |S|$ and $\psi(S') < \psi(S)$; then there is redundancy in S.

Table 4
Types of redundancy

Type	Description	Pattern
I-1	Rules having the same conclusion but different permutations of the same set of conditions	$(RHS(r_i) = RHS(r_j)) \wedge (LHS(r_i) \in \rho(L)) \wedge (LHS(r_j) \in \rho(L))$. where L is a set of literals
I-2	A rule r_k which can be deduced from a set of rules	$\{r_i,....,r_j\} \vdash r_k$, where $(RHS(r_i) = LHS(...)) \wedge ... \wedge (RHS(...) = LHS(r_j)) \wedge (LHS(r_i) = LHS(r_k)) \wedge (RHS(r_j) = RHS(r_k))$
I-3	A rule r which is a specialization of another rule r_k	$(LHS(r_i) \leq LHS(r_j)) \wedge (RHS(r_i) \leq RHS(r_j))$, where $LHS(r_i)$ and $RHS(r_i)$ are specializations based on the same set of substitutions
I-4	A rule r_i which is subsumed by another rule	$(LHS(r_i) \subset LHS(r_j)) \wedge (RHS(r_i) = RHS(r_j))$
I-5	Generalized subsumed rule (r_k is subsumed by r_i and r_j)	$(RHS(r_i) \subset LHS(r_j)) \wedge (RHS(r_i) = RHS(r_k)) \wedge (LHS(r_k) = (LHS(r_i) \cup LHS(r_j) - RHS(r_i)))$
I-6	Rules with same condition(s) and synonymous conclusions	$(RHS(r_i) \cong RHS(r_j)) \wedge (LHS(r_i) = LHS(r_k))$
I-7	Rules with synonymous conditions and same conclusion	$(RHS(r_i) = RHS(r_j)) \wedge (LHS(r_i) \cong LHS(r_k))$
I-8	Rules with synonymous conditions and synonymous conclusion	$(RHS(r_i) \cong RHS(r_j)) \wedge (LHS(r_i) \cong LHS(r_k))$
II-1	Two rules which have the same or synonymous conclusion but contain pair(s) of conflict literals in their conditions	$((RHS(r_i) = RHS(r_j)) \vee (RHS(r_i) \cong RHS(r_j))) \wedge (LHS(r_i) = L \cup \{P\}) \wedge (LHS(r_j) = L \cup \{Q\})$, where L is set of literals and $P \; \hat{\cdot} \; Q$
II-2	A rule with redundant condition(s)	$(P \in LHS(r_i)) \wedge (P' \in LHS(r_i)) \wedge ((P = P') \vee (P \cong P') \vee (P \leq P'))$
II-3	Two rules sharing the same conclusion. and one rule having a singleton condition that is in conflict with a condition of another rule	$(RHS(r_i) = RHS(r_j)) \wedge (LHS(r_i) = L \cup \{P\}) \wedge (LHS(r_j) = \{Q\})$. where L is set of literals and $P \; \hat{\cdot} \; Q$

4.2. Commonly found types of redundancy

If either of the conditions in Definition 15 holds for a given RB. then the RB is said to contain redundancy. Thus. in essence. all types of redundancy are captured by Definition 15. However. in practice. there are sets of commonly found types of redundancy. What are included in Table 4 are the frequently encountered types of redundancy. Type I redundancy in Table 4 involves redundant rule(s) and Type II involves redundant (or unnecessary) literal(s). Each type encompasses a set of specific cases.

4.3. Analysis

Given a set S of rules. $S \vdash C$ indicates the set C of conclusions derivable from S. If we can construct a set S' of rules from S such that Property (a) in Definition 15 is satisfied. we further divide C into C' and C'' where $S' \vdash C'$ and $\Delta \vdash C''$. We can prove that if $S' \approx S$. then $S' \models \Delta$. According to Theorem 1. for every rule $P \in \Delta$. $S' \rightarrow P$ is valid. thus $C'' \subset C'$ and $C = C'$. Therefore. rules in Δ are redundant. During the analysis process we can select a model ζ for S' with regard to the enabling facts and obtain C' from S'. and then obtain C'' from Δ to show $C'' \subset C'$.

When S' is constructed with Property (b) of Definition 15. the number of literals in S' is reduced. even though the number of rules remain the same. Similar analysis can be carried out to prove that $C = C'$. Since S' either contains fewer rules or has fewer literals. we can use S' to replace S. Examples 4 and 5 are used to demonstrate the analysis process.

Example 4. Given the following set S of rules

r_1:	$P \wedge Q \rightarrow R$
r_2:	$A \wedge B \rightarrow U$
r_3:	$U \wedge V \rightarrow W$
r_4:	$R \wedge W \rightarrow D$
r_5:	$P \wedge Q \wedge A \wedge B \wedge V \rightarrow D$

Let $S' = S - \{r_5\}$. We can show that $S' \approx S$ and r_5 is redundant.

Proof. We first convert S and S' into the abbreviated format:

$$S = \{\neg P \neg QR. \neg A \neg BU. \neg U \neg VW,$$
$$\neg R \neg WD. \neg P \neg Q \neg A \neg B \neg VD\}$$

$$S' = \{\neg P \neg QR. \neg A \neg BU. \neg U \neg VW. \neg R \neg WD\}$$

Let ζ be an interpretation. Two situations need to be considered:

1. If $\models_\zeta S$. then $\models_\zeta S'$ is obvious. This is a trivial case.
2. If $\models_\zeta S'$. we need to show that $\models_\zeta S$ also holds. This boils down to proving that $\models_\zeta r_5$. Since $\models_\zeta r_4$ in S'. we must have $\models_\zeta D$ or $\models_\zeta \neg R$ or $\models_\zeta \neg W$

   Case 1: If $\models_\zeta D$. then $\models_\zeta r_5$;
   Case 2: If $\models_\zeta \neg R$. then $\models_\zeta \neg P$ or $\models_\zeta \neg Q$ because $\models_\zeta r_1$. Hence. $\models_\zeta r_5$;

285

Case 3: If $\models_\zeta \neg W$, then $\models_\zeta \neg U$ or $\models_\zeta \neg V$ because $\models_\zeta r_3$.

if $\models_\zeta \neg V$, then $\models_\zeta r_5$.
if $\models_\zeta \neg U$, then either $\models_\zeta \neg A$ or $\models_\zeta \neg B$ because $\models_\zeta r_1$. Thus, $\models_\zeta r_5$.

Therefore, if S' is satisfied under ζ, so is S. $S' \approx S$.

If we choose a model ζ_0 for S' in which $\models_{\zeta_0} \{P, Q, A, B, V\}$, ζ_0 is also a model for r_5. The set of derivable facts from S' and r_5 are $C' = \{R, U, W, D\}$ and $C'' = \{D\}$, respectively. Obviously, $C'' \subseteq C'$, therefore r_5 is redundant. $\square$

S is of redundancy type of I-5. Removing r_5 will eliminate the redundancy.

Example 5. Given the following set S of rules

r_1: $\quad P \wedge Q \wedge W \to R$
r_2: $\quad \neg Q \to R$

Let ϕ_1 be a transformation that results in a rule r_1' by eliminating the literal Q from r_1, and let $S' = \{r_1', r_2\}$. We can show that $S' \approx S$ and the literal Q is redundant (or unnecessary).

Proof. We first convert S and S' into the format below:

$$S = \{\neg P \neg Q \neg WR, QR\}, \qquad S' = \{\neg P \neg WR, QR\}$$

Let ζ be an interpretation. Two cases need to be considered:

1. If $\models_\zeta S'$, then $\models_\zeta S$ is trivial.
2. If $\models_\zeta S$, we need to show that $\models_\zeta S'$ also holds. This boils down to proving that whenever S is satisfied by ζ, $\models_\zeta r_1'$. Since $\models_\zeta r_2$ in S, we must have $\models_\zeta Q$ or $\models_\zeta R$

   Case 1: If $\models_\zeta R$, then $\models_\zeta r_1'$:
   Case 2: If $\models_\zeta Q$ and $\not\models_\zeta R$,[10] then $\models_\zeta \neg P$ or $\models_\zeta \neg W$ must be *true* because $\models_\zeta r_1$. Hence, $\models_\zeta r_1'$.

Therefore, $S' \approx S$, the literal Q in r_1 is redundant. $\square$

S is of redundancy type of II-3. Correcting Type II redundancy involves removing the literal(s) in question. For instance, for Type II-3, when RB contains a rule set S matching the pattern, it can be replaced by the

---

[10] $\not\models_\zeta R$ indicates that R evaluates to false under ζ.

corresponding rule set S' as shown below:

S: $\quad r_1$: $P1 \wedge \cdots \wedge Pk \wedge Q \to R \quad k \geq 1$
$\quad\quad r_2$: $\neg Q \to R$
S': $\quad r_1'$: $P1 \wedge \cdots \wedge Pk \to R$
$\quad\quad r_2$: $\neg Q \to R$

5. KB circularity

5.1. Definition of circularity

Circularity in a KB has been informally defined as a set of rules forming a cycle [7,24,30]. What exactly a circularity entails semantically is not that clear in the literature. In this section, we provide a definition of the KB circularity in terms of the derivation of tautologous rules and argue that the phenomenon reflects an anomalous situation in a KB and has both operational and semantic ramifications.

Definition 16. A rule E is *tautologous*, denoted as $\ddot{E}$, if it contains a complementary or an incompatible pair of literals.

Example 6. Following are two tautologous rules:

- $P \wedge Q \to P$, where $\neg P$ and P are a complementary pair (in $\neg P \vee \neg Q \vee P$)
- $high\_priced(x) \wedge spacious(x) \to expensive(x)$, where $\neg high\_priced(x)$ and $expensive(x)$ are an incompatible pair (in $\neg high\_priced(x) \vee \neg spacious(x) \vee expensive(x)$).

Definition 17. A nonempty set S of rules is *circular* if we can deduce a tautologous rule from S.

Definition 18. A nonempty set S of rules is *minimally circular*, denoted as $\check{S}$, if S is circular and no proper subset of S is circular.

Given $\check{S}$, rules in $\check{S}$ are said to be forming a cycle. The deduction of a tautologous rule is trivial if $\check{S}$ is a singleton set satisfying the aforementioned condition. In a given $\check{S}$, there may be more than one tautologous rule deducible from it that involves different pairs of (complementary or incompatible) literals.

Operationally speaking, circular rules may result in infinite loops (if an exiting condition is not properly defined) during inference, thus hampering the problem solving process. Semantically speaking, the fact that a tautologous

286

wff is derivable indicates that the circular rule set encompasses knowledge that is always true regardless of any problem specific information. In general. tautologous wffs are those that are true by virtue of their logical form and thus provide no useful information about the domain being described [47]. Therefore. circular rules prove to be less useful in the problem solving process. What is needed. as evidenced in many real KB systems. are consistent rules that are triggered by problem specific information (facts) rather than tautologous rules that are true regardless of the problem to be solved.

5.2. Types of circularity

Circularity primarily stems from the definitions of rules in RB. However. control strategies deployed (in places such as the mechanisms of agendas. rule salience or priority level definitions and module selections) in the inference system may also be cause for the infinite looping of certain rules. In this paper. we focus on the types of circularity that are confined in the RB.

Definition 19. Given a minimally circular rule set $\check{S}$. we define two sets of literals $\check{S}_L$ and $\check{S}_R$ as follows:

$$\check{S}_L = \{L | L \in LHS(r) \wedge r \in \check{S}\}$$

$$\check{S}_R = \{L | L \in RHS(r) \wedge r \in \check{S}\}.$$

The types of circularity in a rule base. as summarized in Table 5. are classified based on enumerating possible relationships between $\check{S}_L$ and $\check{S}_R$ and the nature of the tautology. Type I circularity indicates cycles in which $\check{S}_L = \check{S}_R$. Type II describes cycles with additional conditions involved in the rules. therefore. $\check{S}_R$ is a proper subset of $\check{S}_L$. If $C_{\check{S}}$ is a cycle formed out of a minimally circular rule set $\check{S}$. the *girth* g of $C_{\check{S}}$ can be defined as $g(C_{\check{S}}) = |\check{S}|$. Cycles in these types can have a girth ranging from one to some integer MAX where MAX is bounded by the cardinality of the rule base |RB| of a given KB.

5.3. Analysis

The analysis of KB circularity amounts to deriving from a given rule base a tautologous rule r that satisfies the conditions in Definition 16. using some inference method.

Example 7. Below is a rule base S containing five rules

r_1:	$W \rightarrow U$
r_2:	$P \wedge A \rightarrow R$
r_3:	$Q \wedge C \rightarrow W$
r_4:	$R \wedge B \rightarrow Q$
r_5:	$U \wedge D \wedge E \wedge G \rightarrow P$

Using the *resolution* method. we can derive a tautologous rule from S. Since S is the smallest set that yields such a tautologous rule. it is thus minimally circular.

Proof. We convert S into the following format

$$S = \{\neg WU. \neg P \neg AR. \neg Q \neg CW. \neg R \neg BQ.$$

$$\neg U \neg D \neg E \neg GP\}.$$

It is not difficult to see that the following rule is derivable from S by using the resolution method

$$\neg W \neg D \neg E \neg G \neg A \neg B \neg CW.$$

Since $\neg W$ and W are a pair of complementary literals. the derived rule is tautologous. Therefore. S is minimally circular. $\square$

Incidentally. there are four other tautologous rules involving $\neg P$ and P. $\neg Q$ and Q. $\neg R$ and R. and $\neg U$ and U. respectively. This example exhibits Type II-1 circularity.

Once a circularity is detected. the circular rule set needs to be syntactically redefined to break up the circularity. Semantically. information about a problem domain needs to be reorganized so that it will contribute to the problem solving process. Some of the possible remedial measures for circularity can be found in Section 7.

6. KB incompleteness

Informally speaking. a KB is incomplete when it does not have all the necessary information to answer a question of interest in an intended application [16.31]. Thus. completeness represents a query-centric measure for the quality of a KB. KB incompleteness is a real issue to be reckoned with for at least the following reasons: (a) In many applications. the KB is built in an incremental and piecemeal fashion and it undergoes a continual evolution. The information acquired at each stage of the evolution may be vague or indefinite in nature. (b) The deployment of a KB system cannot just wait for the KB to be stabilized in some final and complete form since this may never happen.

Despite the fact that a practical KB can never completely capture all aspects of a real problem domain. it is still possible for a KB to be complete for a specific area in the domain. The boundaries of this specific area may be defined in terms of all *relevant queries* to be asked during problem solving process. If a KB has all the information to answer those relevant queries *definitely*. then the KB is complete with regard

Table 5
Types of circularity in a rule base

Type	Description	Pattern
I-1	$\check{S}_L = \check{S}_R$ for $\check{S}$ and tautologous rule involves complementary pair of literals	$(\check{S}_L = \check{S}_R) \wedge (\check{S} \vdash \check{E}) \wedge (L, \neg L \in \check{E}) \wedge (L \# \neg L)$
I-2	$\check{S}_L = \check{S}_R$ for $\check{S}$ and tautologous rule involves pair of incompatible literals	$(\check{S}_L = \check{S}_R) \wedge (\check{S} \vdash \check{E}) \wedge (L, \neg L \in \check{E}) \wedge (L \cong \neg L)$
II-1	$\check{S}_R \subset \check{S}_L$ for $\check{S}$ and tautologous rule involves complementary pair of literals	$(\check{S}_R \subset \check{S}_L) \wedge (\check{S} \vdash \check{E}) \wedge (L, \neg L \in \check{E}) \wedge (L \# \neg L)$
II-2	$\check{S}_R \subset \check{S}_L$ for $\check{S}$ and tautologous rule involves pair of incompatible literals	$(\check{S}_R \subset \check{S}_L) \wedge (\check{S} \vdash \check{E}) \wedge (L, \neg L \in \check{E}) \wedge (L \cong \neg L)$

to those queries. In what follows, we base our discussions of completeness on the concepts of relevant queries and the ability of a KB to answer those queries.

6.1. Definition of query-based incompleteness

Definition 20. Given a KB, we define $\mathbb{P}_{KB}$ and $\mathbb{P}_A$ as sets of all predicate symbols and *askable* predicate symbols in the KB, respectively. An askable predicate symbol is one that can appear in a query. Usually it is the case that $\mathbb{P}_{KB} \supseteq \mathbb{P}_A$.[11] A query $\bar{Q}$ containing predicate symbols $p_i, \ldots, p_j \in \mathbb{P}_A$ is denoted as

$$\bar{Q} = Q(p_i, \ldots, p_j)$$[12]

Definition 21. A set $\mathbb{Q}$ of *relevant queries* is now defined as follows:

$$\mathbb{Q} = \{\bar{Q}|\bar{Q} \text{ appears in some query session} \wedge$$

$$\bar{Q} = Q(p_i, \ldots, p_j) \wedge p_i, \ldots, p_j \in \mathbb{P}_A\}.$$

Definition 22. Given a query $\bar{Q} \in \mathbb{Q}$, the answer to $\bar{Q}$, denoted as $\alpha(\bar{Q})$, can be either *definite* or *unknown*. $\alpha(\bar{Q})$ is definite if either $KB \vdash \bar{Q}$ or $KB \vdash \neg\bar{Q}$; $\alpha(\bar{Q})$ is unknown if neither $KB \vdash \bar{Q}$ nor $KB \vdash \neg\bar{Q}$.

Definition 23. A KB is *complete* with regard to a relevant query set $\mathbb{Q}$ if $\forall\bar{Q} \in \mathbb{Q} \ [\alpha(\bar{Q}) \text{ is definite}]$.

6.2. Types of incompleteness

Let $\mathbb{P} = \mathbb{P}_{KB} \cup \mathbb{P}_A$. For a predicate symbol $p \in \mathbb{P}$, we

---

[11] When there is incompleteness in a KB, this may not be true, as evidenced in Table 6.

[12] We assume that the query $\bar{Q}$ is a conjunction of the literals containing predicate symbols $P_i, \ldots, P_j$.

introduce a set of predicate symbols $\Re(p)$ on which p directly or indirectly depends. $\Re(p)$ can be obtained using the following procedure.

> INPUT: $p \in \mathbb{P}$
> OUTPUT: $\Re(p)$
> $\Re(p) := \varnothing$;
> **while** $\exists r \in KB \ [p \in RHS(r)]$ **do** $\Re(p) := \Re(p) \cup LHS(r)$;
> **while** $\exists r \in KB \ \exists q \in \mathbb{P}_{KB} \ [q \in RHS(r) \wedge q \in \Re(p) \wedge LHS(r) \not\subseteq \Re(p)]$ **do** $\Re(p) := \Re(p) \cup LHS(r)$;

If a literal containing a predicate symbol p cannot be satisfied by either a given fact or a derived fact, then it is denoted as $\nvdash p$. Three types of incompleteness are defined in Table 6. Types I and II reveal KB incompleteness from the perspective of relevant queries, i.e., lack of necessary information to answer queries, and Type III indicates the potential incompleteness of the relevant query set $\mathbb{Q}$ from the perspective of known information (rules/facts).

Though the classification in Table 6 is exhaustive with regard to Definition 23, there are pragmatic and application specific considerations that will help determine the validity of incompleteness cases.

6.3. Analysis

The analysis of KB incompleteness depends critically on the availability of information regarding the relevant query set in a problem domain. Prototyping often serves as a means to ascertain the relevant query set. If the relevant query set is available, the analysis amounts to finding out if all queries can be answered definitely. Checking for the presence or absence of the aforementioned syntactic symptoms is an integral and necessary part of the analysis process. However, there are other considerations in the analysis process that are semantic, pragmatic, or problem specific. The analysis process is really an iterative one, because as KB continually evolves, so will the relevant query set.

Table 6
Types of incompleteness

Type	Descriptions [7,24,37]	Pattern
I	Dangling conditions, unreachable conclusions	$\exists q \in \mathbb{P} \ \exists p \in \mathbb{P}_A [q \in \Re(p) \wedge \nvdash q]^a$
II	Missing initial facts, missing rules	$\exists p \in \mathbb{P}_A [\Re(p) = \varnothing \wedge p \in \mathbb{P}_{KB}]$
III	Useless conclusions, unused initial facts, isolated rules	$\exists q \in \mathbb{P}_{KB} \ \forall p \in \mathbb{P}_A [q \in \Re(p)]$

^{a} Because the criterion for the completeness issue is domain-specific, it is possible that q in $[q \in \Re(p) \wedge \nvdash q]$ may be useless structure in the KB. Ultimately, the domain expert or knowledge engineer has to determine the nature of the anomaly.

Example 8. For the following KB,

$r_1 : h(x,y) \wedge r(y,z) \rightarrow p_1(x,z) \qquad f_1 : m(d)$

$r_2 : w(y) \wedge u(x) \rightarrow r(x,y) \qquad f_2 : v(a)$

$r_3 : v(x) \rightarrow w(x) \qquad f_3 : u(b)$

$r_4 : m(x) \rightarrow p_3(x) \qquad f_4 : u(c)$

we have

$\mathbb{P}_A = \{p_1, p_2\}$
$\mathbb{P}_{KB} = \{p_1, p_3, r, u, v, w, h, m\}$
$\Re(p_1) = \{h, r, u, v, w\}$
$\Re(p_2) = \varnothing$.

Since $p_2 \in \mathbb{P}_A$ and $[\Re(p_2) = \varnothing \wedge p_2 \notin \mathbb{P}_{KB}]$, there exists Type II incompleteness. No rules and facts could be used to answer queries involving p_2. In addition, $h \in \Re(p_1)$ and $\nvdash h$. So Type I incompleteness also exists. Finally, the presence of the rule r_4 and the fact f_1 may indicate that p_3 should have been an askable predicate. In other words, $\mathbb{P}_A$ is incomplete, and there is reason to believe that the relevant query set is incomplete also. □

7. Remedial measures

Once KB anomalies are identified, the next issue is how to correct the situations in which the quality of a KB has been compromised. Though it is of pivotal importance, the issue has not been adequately addressed in the literature. To a certain extent, this is due to the fact that the issue of how to mend a KB relies on a whole host of considerations, many of which are problem or application specific. In the rest of this section, we would like to address the issue in terms of some general principles and provide some example remedial measures for the cases dealt with in the previous four sections.

For correcting inconsistency, we suggest the following actions:

- Avoid using synonymous literals if possible.
- Delete one of the offending rules that derives the conflict conclusion.
- Modify the conditions (e.g. predicate symbols) of the rules involved such that they no longer have or share the same or synonymous conditions.
- Modify the conclusions (e.g. predicate symbols) of the rules involved such that they are no longer in conflict.
- Move one of the offending rules to a different rule module such that the derivation of conflict conclusions cannot take place in the same problem-solving session or at the same time.

Actions to eliminate redundancy may include:

- Delete redundant rule(s).
- Merge or collapse rules into one.

  For example, $P \wedge Q \rightarrow R$, $\neg P \wedge Q \rightarrow R \Rightarrow Q \rightarrow R$

- Delete condition(s) of certain rule(s).

  For example, $P \wedge Q \rightarrow R$, $\neg Q \rightarrow R \Rightarrow P \rightarrow R$, $\neg Q \rightarrow R$

- Modify the conditions or conclusions of the redundant rules such that they no longer are the same or synonymous.

To resolve circularity, the following remedial measures may be taken:

- Remove a rule from a circular rule set.

  For example, $P \rightarrow Q$, $Q \rightarrow R$, $R \rightarrow P \Rightarrow P \rightarrow Q$, $Q \rightarrow R$

- Redefine a conclusion of a rule in the set such that it no longer serves as a condition of another rule in the set.

  For example, $P \rightarrow Q$, $Q \rightarrow R$, $R \rightarrow P \Rightarrow P \rightarrow Q$, $Q \rightarrow R'$, $R \rightarrow P$ where R' and R are no longer unifiable.

- Redefine a condition of a rule in the set such that it no longer matches a conclusion of another rule in the set.

To plug holes in an incomplete KB, we could

- Add new rules and/or facts to make all relevant queries definite.

  For example, new rules and facts can be added to make $h(x, y)$ satisfiable in Example 8.

  - Modify the initial facts to patch up holes.
  - Modify the conditions and/or conclusions of rules involved in an incompleteness case so that they will be "connected" with the rest of RB.

Though it is beyond the scope of this paper, we would like to point out that in a KB where certainty factors (CF) are used, there are additional actions to be considered. For instance, add or modify CF values for rules or facts, or

modify the threshold value(s) for the CF-value propagation during inference process.

8. Concluding remarks

As more and more expert systems and knowledge-based systems are deployed in settings where failures may result in loss of productivity. decision-making quality. property. business. services. investment. or even life. ways to detect and resolve potential anomalies in a KB become critical issues in developing correct. accurate and reliable systems. In order for the results to be credible. V&V techniques must be built on a solid theoretical foundation.

It is difficult to assess many of the V&V tools. methods and techniques that have been developed or proposed because there is no accepted standard against which to measure the reliability or correctness of an expert system. Indeed there is lack of definite semantics for expert systems in general and KB in particular. This prevents any definite conclusions about reliability and hinders the use of expert systems in safety-critical applications. The field of V&V for expert systems is far from having tractable formal models that can cover all of the features of real expert systems. which often rely on imperative state changes and other non-logical features. Our simplified model. though a preliminary one. does provide a basis for reaching definite conclusions about the reliability of those aspects in expert systems that can be expressed in logical terms. It is our hope that the logical formulation presented in this paper makes a step in the right direction.

Future work can continue in several directions. One is concerned with how to establish an assessment standard, based on logical instruments similar to those discussed in this paper. for the V&V tools and methodologies. For instance. given a KB and its semantics Γ, we use $\tilde{A}_\Gamma$ to indicate the set of anomalies defined under Γ. For a V&V method M. we use $\tilde{A}_M$ to denote the set of anomalies M is capable of discovering. M is *sound* if $\forall \tilde{a} \in \tilde{A}_M [\tilde{a} \in \tilde{A}_\Gamma]$; M is *complete* if $\forall \tilde{a} \in \tilde{A}_\Gamma [\tilde{a} \in \tilde{A}_M]$.

Another direction is to study the KB anomalies in an object-oriented (OO) paradigm. Recent developments in knowledge representation formalisms include: (a) extending the OO paradigm to include rules (i.e. rules can be considered as a specific type of behavior for objects); (b) bringing objects into the rule-based paradigm (i.e. rules are specified about objects); (c) hybrid representation formalism that blends frames. objects. cases and rules together [48]. The next challenge to us is the issue of how to (re)define the concepts and meanings of KB anomalies in the context of those formalisms.

A KB should be developed based on its underlying ontology [49.50]. It is not clear what relationship there is between the anonymous situations that are manifested at a KB level and the root causes at its ontology. This is yet another direction worth exploring.

References

[1] M. Ayel. J.P. Laurent (Eds.). Validation. Verification and Test of Knowledge-Based Systems Wiley. Chichester. UK. 1991.
[2] U.G. Gupta (Ed.). Validating and Verifying Knowledge-Based Systems IEEE Computer Society Press. Los Alamitos. CA. 1991.
[3] Verification and validation of knowledge-based systems [Special issue]. C. Culbert (Ed.). Expert Systems with Applications 1 (3) (1990).
[4] Verification and validation of intelligent systems: five years of AAAI workshops [Special issue]. D.E. O'Leary (Ed.). International Journal of Intelligent Systems 9 (8/9) (1994).
[5] E. Plaza. Validation and verification of knowledge-based systems. IEEE Expert 8 (3) (1993) 45–81.
[6] Verification and validation of knowledge-based systems. A. Preece. C. Suen (Eds.). International Journal of Expert Systems 6 (2/3) (1993) (special issue).
[7] C.L. Chang. J.B. Combs. R.A. Stachowitz. A report on the expert systems validation associate (EVA). Expert Systems with Applications 1 (1990) 217–230.
[8] B.J. Cragun. H.J. Steudel. A decision-table-based processor for checking completeness and consistency in rule-based expert systems. International Journal of Man–Machine Studies 26 (1987) 633–648.
[9] R.F. Gamble. G.C. Roman. W.E. Ball. Formal verification of pure production system programs. Proceedings of Ninth National Conference on AI. 1991. pp. 329–334.
[10] A. Ginsberg. Knowledge-base reduction: a new approach to checking knowledge bases for inconsistency and redundancy. Proceedings of Seventh National Conference on AI. 1998. pp. 585–589.
[11] A. Ginsberg. K. Williamson. Inconsistency and rdundancy checking for quasi-first-order-logic knowledge bases. International Journal of Expert Systems 6 (3) (1993) 321–340.
[12] D. Hamilton. K. Kelley. C. Culbert. State-of-the-practice in knowledge-based system verification and validation. Expert Systems with Applications 3 (1991) 403–410.
[13] M. Jafar. A.T. Bahill. Interactive verification of knowledge-based systems. IEEE Expert 8 (1) (1993) 25–32.
[14] J.D. Kiper. Structural testing of rule-based expert systems. ACM Transactions on Software Engineering and Methodology 1 (2) (1992) 168–187.
[15] S. Kirani. I.A. Zaulkeman. W.T. Tsai. Evaluation of expert system testing methods. Communications of ACM 37 (11) (1994) 71–81.
[16] H.J. Levesque. The logic of incomplete knowledge bases. in: M.L. Brodie. J. Mylopoulos. J.W. Schmidt (Eds.). On Conceptual Modeling. Springer. New York. 1984.
[17] N.K. Liu. T. Dillon. An approach towards the verification of expert systems using numerical petri nets. International Journal of Intelligent Systems 6 (1991) 255–276.
[18] Luqi. D. Cooke. How to combine nonmonotonic logic and rapid prototyping to help maintain software. International Journal of Software Engineering and Knowledge Engineering (1999).
[19] Luqi. Knowledge-based support for rapid software prototyping. IEEE Expert 3 (4) (1988) 9–18.
[20] Luqi. Rapid prototyping languages and expert systems. IEEE Expert 4 (2) (1989) 2–5.
[21] L.A. Miller. Recommended guidelines for V&V of cases. AAAI-94

Workshop on Validation and Verification of Knowledge-Based Systems. 1994, pp. 1–9.

[22] T.J. Murray, M.R. Tanniru. Control of inconsistency and redundancy in prolog-type knowledge bases. Expert Systems with Applications 2 (1991) 321–331.

[23] D.L. Nazareth. Investigating the applicability of petri nets for rule-based system verification. IEEE Transactions on Knowledge and Data Engineering 5 (3) (1993) 402–415.

[24] T.A. Nguyen. W.A. Perkins, T.J. Laffey, D. Pecora. Knowledge base verification. AI Magazine 8 (1987) 69–75.

[25] H. Nonfjall, H.L. Larsen. Detection of potential inconsistencies in knowledge bases. International Journal of Intelligent Systems 7 (1992) 81–96.

[26] R.M. O'Keefe, D.E. O'Leary. Expert system verification and validation: a survey and tutorial. Artificial Intelligence Review 7 (1993) 3–42.

[27] R. Plant, S. Murrell. On the validation and verification of production systems: a graph reduction approach. AAAI-94 Workshop on Validation and Verification of Knowledge-Based Systems. 1994, pp. 56–63.

[28] A. Preece, R. Shinghal, A. Batarekh. Verifying expert systems: a logical framework and a practical tool. Expert Systems with Applications 5 (2/3) (1992) 421–436.

[29] M.C. Rousset. On the consistency of knowledge bases: the COVADIS system. Proceedings of Eighth European Conference on AI. 1988, pp. 79–84.

[30] J. Rushby. Quality Measures and Assurance for AI Software. NASA Contractor Report 4187, October 1988.

[31] J. Rushby, R.A. Whitehurst. Formal Verification of AI Software. NASA Contractor Report 181827, February 1989.

[32] M. Suwa, A.C. Scott, E.H. Shortliffe. An approach to verifying completeness and consistency in a rule-based expert system. AI Magazine 3 (1982) 16–21.

[33] W.T. Tsai, I.A. Zualkernan, S. Kirani. Pragmatic testing methods for expert systems. International Journal of AI Tools 2 (2) (1993) 181–217.

[34] Verifying Knowledge Bases: A Bibliography. Knowledge Engineering Review 7 (2) (1992) 143–146.

[35] R.J. Waldinger, M. Stickel. Proving properties of rule-based systems. International Journal of Software Engineering and Knowledge Engineering 2 (1) (1992) 121–144.

[36] D. Zhang, Luqi. Formal analysis of inconsistency and redundancy in knowledge bases. IJCAI-95 Workshop on Verification and Validation of Knowledge-Based Systems, pp. 110–116.

[37] D. Zhang, D. Nguyen. PREPARE: a tool for knowledge base verification. IEEE Transactions on Knowledge and Data Engineering 6 (6) (1994) 983–989.

[38] D. Zhang. Perspectives in knowledge base verification. Proceedings of Fifth International Conference on Software Engineering and Knowledge Engineering. 1993, pp. 123–456.

[39] N. Zlatareva. A framework for verification, validation, and refinement of knowledge bases: the VVR system. International Journal of Intelligent Systems 9 (1994) 703–737.

[40] N. Zlatareva, A. Preece. State of the art in automated validation of knowledge-based systems. Expert Systems with Applications 7 (2) (1994) 151–167.

[41] R. de Hoog, et al.. The CommonKADS organization model: content, usage, and computer support. Expert Systems with Applications 11 (1) (1996) 247–260.

[42] G. Guida, C. Tasso. Design and Development of Knowledge-Based Systems: From Life Cycle to Methodology. Wiley, Chichester, UK. 1994.

[43] G. Schreiber, et al.. CommonKADS: a comprehensive methodology for KBS development. IEEE Expert 9 (6) (1994) 28–37.

[44] D.S.W. Tansley, C.C. Hayball. Knowledge-Based Systems Analysis and Design: A KADS Developer's Handbook. Prentice Hall, UK. 1993.

[45] M. Ben-Ari. Mathematical Logic for Computer Science. Prentice Hall, UK. 1993.

[46] C.L. Chang, R.C.T. Lee. Symbolic Logic and Mechanical Theorem Proving. Academic Press, New York. 1973.

[47] M.R. Genesereth, N.J. Nilsson. Logical Foundations of Artificial Intelligence. Morgan Kaufmann Publishers, Los Altos, CA. 1987.

[48] K.W. Tracy, P. Bouthhoorn. Object-Oriented Artificial Intelligence Using C++. Computer Science Press, New York. 1997.

[49] B. Chandrasekaran, J.R. Josephson, V.R. Benjamins. What are ontologies, and why do we need them? IEEE Intelligent Systems 14 (1) (1999) 20–26.

[50] D.E. O'Leary. Using AI in knowledge management: knowledge bases and ontologies. IEEE Intelligent Systems 13 (3) (1998) 34–39.

APPLYING MACHINE LEARNING ALGORITHMS
IN SOFTWARE DEVELOPMENT

Du Zhang

Department of Computer Science
California State University
Sacramento, CA 95819-6021
zhangd@ecs.csus.edu

Abstract

Machine learning deals with the issue of how to build programs that improve their performance at some task through experience. Machine learning algorithms have proven to be of great practical value in a variety of application domains. They are particularly useful for (a) poorly understood problem domains where little knowledge exists for the humans to develop effective algorithms; (b) domains where there are large databases containing valuable implicit regularities to be discovered; or (c) domains where programs must adapt to changing conditions. Not surprisingly, the field of software engineering turns out to be a fertile ground where many software development tasks could be formulated as learning problems and approached in terms of learning algorithms. In this paper, we first take a look at the characteristics and applicability of some frequently utilized machine learning algorithms. We then provide formulations of some software development tasks using learning algorithms. Finally, a brief summary is given of the existing work.

Keywords: machine learning, software engineering, learning algorithms.

1. The Challenge

The challenge of modeling software system structures in a fastly moving scenario gives rise to a number of demanding situations. First situation is where software systems must dynamically adapt to changing conditions. The second one is where the domains involved may be poorly understood. And the last but not the least is one where there may be no knowledge (though there may be raw data available) to develop effective algorithmic solutions.

To answer the challenge, a number of approaches can be utilized [1,12]. One such approach is the *transformational programming*. Under the transformational programming, software is developed, modified, and maintained at specification level, and then automatically transformed into production-quality software through automatic program synthesis [5]. This software development paradigm will enable software engineering to become the discipline of capturing and automating currently undocumented domain and design knowledge [10]. Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs.

In order to realize its full potential, there are tools and methodologies needed for the various tasks inherent to the transformational programming. In this paper, we take a look at how machine learning (ML) algorithms can be used to build tools for software development and maintenance tasks. The rest of the paper is organized as follows. Section 2 provides an overview of machine learning and frequently used learning algorithms. Some of the software development and maintenance tasks for which learning algorithms are applicable are given in Section 3. Formulations of those tasks in terms of the learning

algorithms are discussed in Section 4. Section 5 describes some of the existing work. Finally in Section 6, we conclude the paper with remarks on future work.

2. Machine Learning Algorithms

Machine learning deals with the issue of how to build computer programs that improve their performance at some task through experience [11]. Machine learning algorithms have been utilized in: (1) data mining problems where large databases may contain valuable implicit regularities that can be discovered automatically; (2) poorly understood domains where humans might not have the knowledge needed to develop effective algorithms; and (3) domains where programs must dynamically adapt to changing conditions [11]. Learning a target function from training data involves many issues (function representation, how and when to generate the function, with what given input, how to evaluate the performance of generated function, and so forth). Figure 1 describes the dimensions of the target function learning.

Major types of learning include: concept learning (CL), decision trees (DT), artificial neural networks (ANN), Bayesian belief networks (BBN), reinforcement learning (RL), genetic algorithms (GA) and genetic programming (GP), instance-based learning (IBL), inductive logic programming (ILP), and analytical learning (AL). Table 1 summarizes the main properties of different types of learning.

Not surprisingly, machine learning methods can be (and some have already been) used in developing better tools or software products. Our preliminary study identifies the software development and maintenance tasks in the following areas to be appropriate for machine learning applications: requirement engineering (knowledge elicitation, prototyping); software reuse (application generators); testing and validation; maintenance (software understanding); project management (cost, effort, or defect prediction or estimation).

3. Software Engineering Tasks

Table 2 contains a list of software engineering tasks for which ML methods are applicable. Those tasks belong to different life-cycle processes of requirement specification, design, implementation, testing and maintenance. This list is by no means a complete one. It only serves as a harbinger of what may become a fertile ground for some exciting research on applying ML techniques in software development and maintenance.

One of the attractive aspects of ML techniques is the fact that they offer an invaluable complement to the existing repertoire of tools so as to make it easier to rise to the challenge of the aforementioned demanding situations.

4. Applying ML Algorithms to SE Tasks

In this section, we formulate the identified software development and maintenance tasks as learning problems and approach the tasks using machine learning algorithms.

Component reuse

Component retrieval from a software repository is an important issue in supporting software reuse. This task can be formulated into an instance-based learning problem as follows:
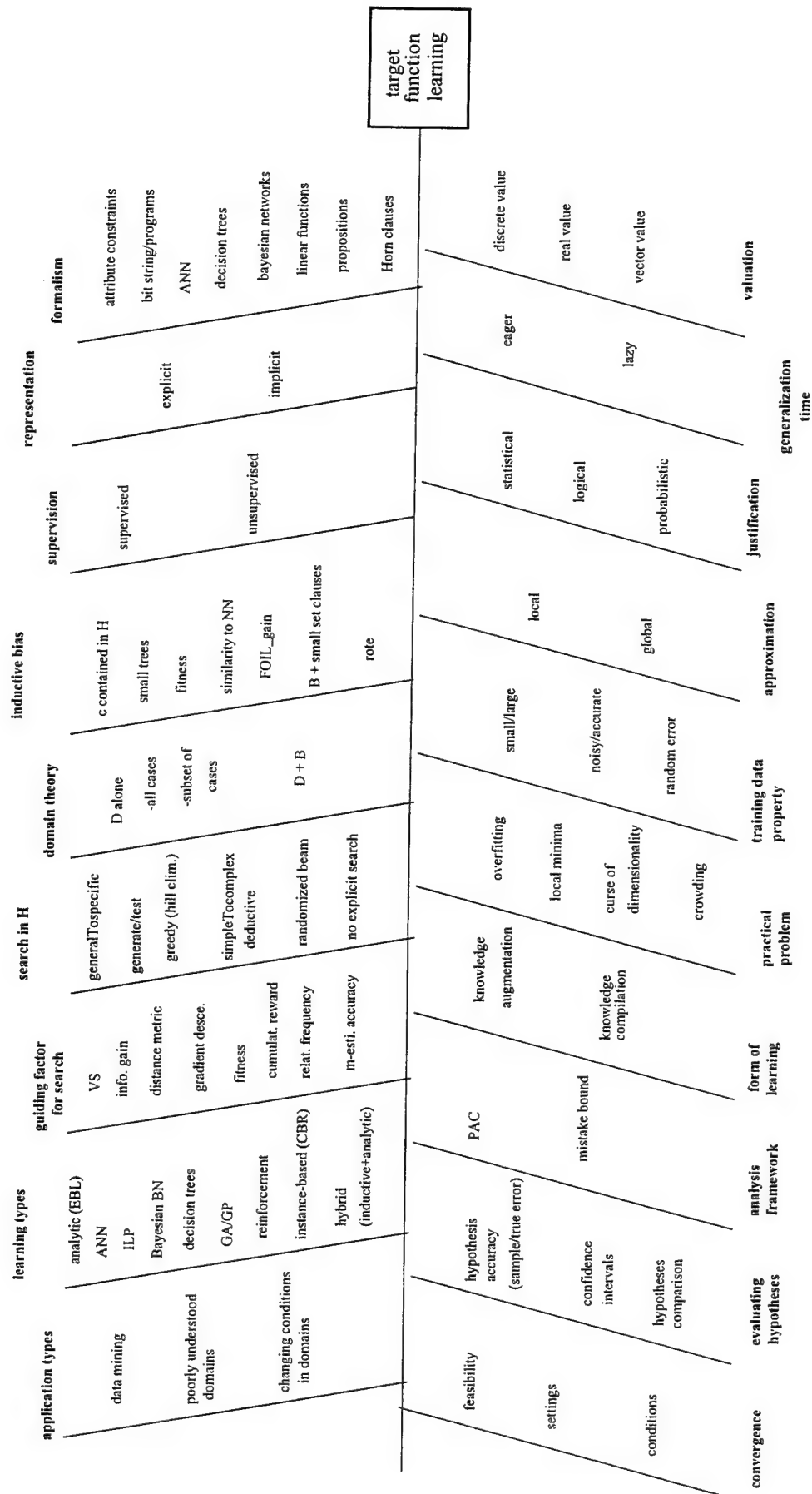
Figure 1. Dimensions of learning.

Table 1. Major types of learning methods[1].

Type	Target function	Target function generation[2]	Search	Inductive bias	Algorithm[3]
AL	Horn clauses	Eager, supervised, D + B	Deductive reasoning	B + set of Horn clauses	Prolog-EBG
ANN	ANN	Eager, supervised, D (global)	Gradient descent guided	Smooth interpolation between data points	Back-propagation
BBN	Bayesian network	Eager, supervised, D (global), explicit or implicit	Probabilistic, no explicit search	Minimum description length	MAP, BOC, Gibbs, NBC
CL	Conjunction of attribute constraints	Eager, supervised, D (global)	Version Space (VS) guided	$c \in H$	Candidate_elimination
DT	Decision trees	Eager, supervised, D (global)	Information gain (entropy)	Preference for small trees	ID3, C4.5, Assistant
GA GP	Bit strings, program trees	Eager, unsupervised, no D	Hill climbing (simulated evolution)	Fitness-driven	Prototypical GA/GP algorithms
IBL	Not explicitly defined	Lazy, supervised, D (local)	Statistical reasoning	Similarity to NN	K-NN, LWR, CBR
ILP	If-then rules	Eager, supervised, D (global)	Statistical, general-to-specific	Rule accuracy, FOIL-gain, shorter clauses	SCA, FOIL, inverse resolution
RL	Control strategy π^*	Eager, unsupervised, no D	Through training episodes	Actions with max. Q value	Q, TD

---

[1] The classification here is based on materials in [11].
[2] The sets D and B refer to training data and domain theory, respectively.
[3] The algorithms listed are only representatives from different types of learning.

Table 2. SE tasks and applicable ML methods.

SE tasks	Applicable type(s) of learning
Requirement engineering	AL, BBN, LL, DT, ILP
Rapid prototyping	GP
Component reuse	IBL (CBR[4])
Cost/effort prediction	IBL (CBR), DT, BBN, ANN
Defect prediction	BBN
Test oracle generation	AL (EBL[5])
Test data adequacy	CL
Validation	AL
Reverse engineering	CL

1. Components in a software repository are represented as points in the n-dimensional Euclidean space (or cases in a case base).
2. Information in a component can be divided into *indexed* and *unindexed* information (attributes). Indexed information is used for retrieval purpose and unindexed information is used for contextual purpose. Because of the *curse of dimensionality* problem [11], the choice of indexed attributes must be judicious.
3. Queries to the repository for desirable components can be represented as constraints on indexable attributes.
4. Similarity measures for the nearest neighbors of the desirable component can be based on the standard Euclidean distance, distance-weighted measure, or symbolic measure.
5. The possible retrieval methods include: *K-Nearest Neighbor, inductive retrieval, Locally Weighted Regression.*
6. The adaptation of the retrieved component for the task at hand can be *structural* (applying adaptation rules directly to the retrieved component), or *derivational* (reusing adaptation rules that generated the original solution to produce a new solution).

Rapid prototyping

Rapid prototyping is an important tool for understanding and validating software requirements. In addition, software prototypes can be used for other purposes such as user training and system testing [18]. Different prototyping techniques have been developed for *evolutionary* and *throw-away* prototypings. The existing techniques can be augmented by including a machine learning approach, i.e., the use of genetic programming.

In GP, a computer program is often represented as a program tree where the internal nodes correspond to a set of functions used in the program and the external nodes (terminals) indicate variables and constants used as input to functions. For a given problem, GP starts with an initial population of randomly generated computer programs. The evolution process of generating a final computer program that solves the given problem hinges on some sort of fitness evaluation and probabilistically reproducing the next generation of the

---

[4] CBR stands for case-based reasoning.
[5] EBL refers to explanation-based learning.

program population through some genetic operations. Given a GP development environment such as the one in [8], the framework of a GP-based rapid prototyping process can be described as follows:

1. Define sets of functions and terminals to be used in the developed (prototype) systems.
2. Define a *fitness* function to be used in evaluating the worthiness of a generated program. Test data (input values and expected output) may be needed in assisting the evaluation.
3. Generate the initial program population.
4. Determine selection strategies for programs in the current generation to be included in the next generation population.
5. Decide how the genetic operations (*crossover* and *mutation*) are carried out during each generation and how often these operations are performed.
6. Specify the terminating criteria for the evolution process and the way of checking for termination.
7. Translate the returned program into a desired programming language format.

Requirement engineering

Requirement engineering refers to the process of establishing the services a system should provide and the constraints under which it must operate [18]. A requirement may be functional or non-functional. A functional requirement describes a system service or function, whereas a non-functional requirement represents a constraint imposed on the system. How to obtain functional requirements of a system is the focus here. The situation in which ML algorithms will be particularly useful is when there exist empirical data from the problem domain that describe how the system should react to certain inputs. Under this circumstance, functional requirements can be "learned" from the data through some learning algorithm.

1. Let X and C be the domain and the co-domain of a system function f to be learned. The data set D is defined as: $D = \{<x_i, c_k>| x_i \in X \wedge c_k \in C\}$.
2. The target functions f to be learned is such that $\forall x_i \in X$ and $\forall c_k \in C, f(x_i) = c_k$.
3. The learning methods applicable here have to be of *supervised* type. Depending on the nature of the data set D, different learning algorithms (in AL, BBN, CL, DT, ILP) can be utilized to capture (learn) a system's functional requirements.

Reverse engineering

Legacy systems are old systems that are critical to the operation of an organization which uses them and that must still be maintained. Most legacy systems were developed before software engineering techniques were widely used. Thus they may be poorly structured and their documentation may be either out-of-date or non-existent. In order to bring to bear the legacy system maintenance, the first task is to recover the design or specification of a legacy system from its source or executable code (hence, the term of reverse engineering, or program comprehension and understanding). Below we describe a framework for deriving functional specification of a legacy software system from its executable code.

1. Given the executable code p and its input data set X, and output set C, the training data set D is defined as: $D = \{<x_i, p(x_i)>| x_i \in X \wedge p(x_i) \in C\}$.
2. The process of deriving the functional specification f for p can be described as a learning problem in which f is learned through some ML algorithm such that
$$\forall x_i \in X [f(x_i) = p(x_i)].$$
3. Many supervised learning methods can be used here (e.g., CL).

Validation

Verification and validation are important checking processes to make sure that implemented software system conforms to its specification. To check a software implementation against its specification, we assume the availability of both a specification and an executable code. This checking process can be performed as an analytic learning task as follows:

1. Let X and C be the domain and co-domain of the implementation (executable code) p, which is defined as: $p: X \rightarrow C$.
2. The training set D is defined as: $D = \{<x_i, p(x_i)>| x_i \in X \}$.
3. The specification for p is denoted as B, which corresponds to the domain theory in the analytic learning.
4. The validation checking is defined to be: p is valid if

$$\forall <x_i, p(x_i)> \in D \ [B \wedge x_i \vdash p(x_i)].$$

5. Explanation-based learning algorithms can be utilized to carry out the checking process.

Test oracle generation

Functional testing involves executing a program under test and examining the output from the program. An oracle is needed in functional testing in order to determine if the output from a program is correct. The oracle can be a human or a software one [13]. The approach we propose here allows a test oracle to be learned as a function from the specification and a small set of training data. The learned test oracle can then be used for the functional testing purpose.

1. Let X and C be the domain and co-domain of the program p to be tested. Let B be the specification for p.
2. Define a small training set D as: $D = \{<x_i, p(x_i)>| x_i \in X' \wedge X' \subset X \wedge p(x_i) \in C\}$.
3. Use the explanation-based learning (EBL) to generate a test oracle Θ ($\Theta: X \rightarrow C$) for p from B and D.
4. Use Θ for the functional testing: $\forall x_i \in X$ [output of p is correct if $p(x_i) = \Theta(x_i)$].

Test adequacy criteria

Software test data adequacy criteria are rules that determine if a software product has been adequately tested [21]. A test data adequacy criterion ζ is a function: $\zeta: P \times S \times T \rightarrow \{true, false\}$ where P is a set of programs, S a set of specifications and T the class of test sets. $\zeta(p, s, t) = true$ means that t is adequate for testing program p against specification s according to criterion ζ. Since ζ is essentially a Boolean function, we can use a strategy such as CL to learn the test data adequacy criteria.

1. Define the instance space X as: $X = \{ <p_i, s_j, t_k>| p_i \in P \wedge s_j \in S \wedge t_k \in T\}$.
2. Define the training data set D as: $D = \{<x, \zeta(x)>| x \in X \wedge \zeta(x) \in V\}$, where V is defined as: $V = \{true, false\}$.
3. Use the concept of *version space* and the *candidate-elimination* algorithm in CL to learn the definition of ζ.

Software defect prediction

Software defect prediction is a very useful and important tool to gauge the likely delivered quality and maintenance effort before software systems are deployed [4]. Predicting defects requires a holistic model rather than a single-issue model that hinges on either size, or complexity, or testing metrics, or process quality data alone. It is argued in [4] that all

298

these factors must be taken into consideration in order for the defect prediction to be successful.

Bayesian Belief Networks (BBN) prove to be a very useful approach to the software defect prediction problem. A BBN represents the *joint probability distribution* for a set of variables. This is accomplished by specifying (a) a directed acyclic graph (DAG) where nodes represent variables and arcs correspond to *conditional independence* assumptions (causal knowledge about the problem domain), and (b) a set of local conditional probability tables (one for each variable) [7, 11]. A BBN can be used to infer the probability distribution for a target variable (e.g., "Defects Detected"), which specifies the probability that the variable will take on each of its possible values (e.g., "very low", "low", "medium", "high", or "very high" for the variable "Defects Detected") given the observed values of the other variables. In general, a BBN can be used to compute the probability distribution for any subset of variables given the values or distributions for any subset of the remaining variables. When using a BBN for a decision support system such as software defect prediction, the steps below should be followed.

1. Identify variables in the BBN. Variables can be: (a) *hypothesis* variables for which the user would like to find out their probability distributions (hypothesis variable are either unobservable or too costly to observe), (b) *information* variables that can be observed, or (c) *mediating* variables that are introduced for certain purpose (help reflect independence properties, facilitate acquisition of conditional probabilities, and so forth). Variables should be defined to reflect the life-cycle activities (specification, design, implementation, and testing) and capture the multi-facet nature of software defects (perspectives from size, testing metrics and process quality). Variables are denoted as nodes in the DAG.
2. Define the proper causal relationships among variables. These relationships also should capture and reflect the causality exhibited in the software life-cycle processes. They will be represented as arcs in the corresponding DAG.
3. Acquire a probability distribution for each variable in the BBN. Theoretically well-founded probabilities, or frequencies, or subjective estimates can all be used in the BBN. The result is a set of conditional probability tables one for each variable. The full joint probability distribution for all the defect-centric variables is embodied in the DAG structure and the set of conditional probability tables.

Project effort (cost) prediction

How to estimate the cost for a software project is a very important issue in the software project management. Most of the existing work is based on algorithmic models of effort [17]. A viable alternative approach to the project effort prediction is instance-based learning. IBL yields very good performance for situations where an algorithmic model for the prediction is not possible. In the framework of IBL, the prediction process can be carried out as follows.

1. Introduce a set of features or attributes (e.g., number of interfaces, size of functional requirements, development tools and methods, and so forth) to characterize projects. The decision on the number of features has to be judicious, as this may become the cause of the *curse of dimensionality* problem that will affect the prediction accuracy.
2. Collect data on completed projects and store them as instances in the case base.
3. Define *similarity* or *distance* between instances in the case base according to the symbolic representations of instances (e.g., Euclidean distance in an n-dimensional space where n is the number of features used). To overcome the potential curse of

dimensionality problem, features may be weighed differently when calculating the distance (or similarity) between two instances.

4. Given a query for predicting the effort of a new project, use an algorithm such as *K-Nearest Neighbor*, or, *Locally Weighted Regression* to retrieve similar projects and use them as the basis for returning the prediction result.

5. Existing Work

Several areas in software development have already witnessed the use of machine learning methods. In this section, we take a look at some reported results. The list is definitely not a complete one. It only serves as an indication that people realize the potential of ML techniques and begin to reap the benefits from applying them in software development and maintenance.

Scenario-based requirement engineering

The work reported in [9] describes a formal method for supporting the process of inferring specifications of system goals and requirements inductively from interaction scenarios provided by stakeholders. The method is based on a learning algorithm that takes scenarios as examples and counter-examples (positive and negative scenarios) and generates goal specifications as temporal rules.

A related work in [6] presents a scenarios-based elicitation and validation assistant that helps requirements engineers acquire and maintain a specification consistent with scenarios provided. The system relies on explanation-based learning (EBL) to generalize scenarios to state and prove validation lemmas.

Software project effort estimation

Instance-based learning techniques are used in [17] for predicting the software project effort for new projects. The empirical results obtained (from nine different industrial data sets totaling 275 projects) indicate that cased-based reasoning offers a viable complement to the existing prediction and estimations techniques. A related CBR application in software effort estimation is given in [20].

Decision trees (DT) and artificial neural networks (ANN) are used in [19] to help predict software development effort. The results were competitive with conventional methods such as COCOMO and function points. The main advantage of DT and ANN based estimation systems is that they are adaptable and nonparametric.

The result reported in [3] indicates that the improved predictive performance can be obtained through the use of Bayesian analysis. Additional research on ML based software effort estimation can be found in [2,14,15,16].

Software defect prediction

Bayesian belief networks are used in [4] to predict software defects. Though the system reported is only a prototype, it shows the potential BBN has in incorporating multiple perspectives on defect prediction into a single, unified model.

Variables in the prototype BBN system [4] are chosen to represent the life-cycle processes of specification, design and implementation, and testing (Problem-Complexity, Design-Effort, Design-Size, Defects-Introduced, Testing-Effort, Defects-Detected, Defects-Density-At-Testing, Residual-Defect-Count, and Residual-Defect-Density). The proper causal relationships among those software life-cycle processes are then captured and reflected as arcs connecting the variables.

A tool is then used with regard to the BBN model in the following manner. For given facts about Design-Effort and Design-Size as input, the tool will use Bayesian inference to derive the probability distributions for Defects-Introduced, Defects-Detected and Defect-Density.

6. Concluding Remarks

In this paper, we show how ML algorithms can be used in tackling software engineering problems. ML algorithms not only can be used to build tools for software development and maintenance tasks, but also can be incorporated into software products to make them adaptive and self-configuring. A maturing software engineering discipline will definitely be able to benefit from the utility of ML techniques.

What lies ahead is the issue of realizing the promise and potential ML techniques have to offer in the circumstances as discussed in Section 4. In addition, expanding the frontier of ML application in software engineering is another direction worth pursuing.

References

1. B. Boehm, "Requirements that handle IKIWISI, COTS, and rapid change," *IEEE Computer*, Vol. 33, No. 7, July 2000, pp.99-102.

2. L. Briand, V. Basili and W. Thomas, "A pattern recognition approach for software engineering data analysis," *IEEE Trans. SE*, Vol. 18, No. 11, November 1992, pp. 931-942.

3. S. Chulani, B. Boehm and B. Steece, "Bayesian analysis of empirical software engineering cost models," *IEEE Trans. SE*, Vol. 25, No. 4, July 1999, pp. 573-583.

4. N. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. SE*, Vol. 25, No. 5, Sept. 1999, pp. 675-689.

5. C. Green et al, "Report on a knowledge-based software assistant, In *Readings in Artificial Intelligence and Software Engineering*, eds. C. Rich and R.C. Waters, Morgan Kaufmann, 1986, pp.377-428.

6. R.J. Hall, "Systematic incremental validation of reactive systems via sound scenario generalization," *Automatic Software Eng.*, Vol.2, pp.131-166, 1995.

7. F.V. Jensen, *An Introduction to Bayesian Networks*, Springer, 1996.

8. M. Kramer, and D. Zhang, "Gaps: a genetic programming system," *Proc. of IEEE International Conference on Computer Software and Applications* (COMPSAC 2000).

9. van Lamsweerde and L. Willemet, "Inferring declarative requirements specification from operational scenarios," *IEEE Trans. SE*, Vol. 24, No. 12, Dec. 1998, pp.1089-1114.

10. M. Lowry, "Software engineering in the twenty first century", *AI Magazine*, Vol.14, No.3, Fall 1992, pp.71-87.

11. T. Mitchell, *Machine Learning*, McGraw-Hill, 1997.

12. D. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. SE*, Vol. 5, No. 3, March 1979, pp. 128-137.

13. D. Peters and D. Parnas, "Using test oracles generated from program documentation," *IEEE Trans. SE*, Vol. 24, No. 3, March 1998, pp. 161-173.

14. A. Porter and R. Selby, "Empirically-guided software development using metric-based classification trees," *IEEE Software*, Vol. 7, March 1990, pp. 46-54.

15. A. Porter and R. Selby, "Evaluating techniques for generating metric-based classification trees," *J. Systems Software*, Vol. 12, July 1990, pp. 209-218.

16. R. Selby and A. Porter, "Learning from examples: generation and evaluation of decision trees for software resource analysis," *IEEE Trans. SE*, Vol. 14, 1988, pp.1743-1757.

17. M. Shepperd and C. Schofield, "Estimating software project effort using analogies", *IEEE Trans. SE*, Vol. 23, No.12, November 1997, pp. 736-743.

18. I. Sommerville, *Software Engineering*, Addison-Wesley, 1996.

19. K. Srinivasan and D. Fisher, "Machine learning approaches to estimating software development effort," *IEEE Trans. SE*, Vol. 21, No. 2, Feb. 1995, pp. 126-137.

20. S. Vicinanza, M.J. Prietulla, and T. Mukhopadhyay, "Case-based reasoning in software effort estimation," *Proc. 11th Int'l. Conf. On Information Systems*, 1990, pp.149-158.

21. H. Zhu, "A formal analysis of the subsume relation between software test adequacy criteria," *IEEE Trans. SE*, Vol.22, No.4, April 1996, pp.248-255.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    8725 John J. Kingman Rd., STE 0944
    Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52   2
    Naval Postgraduate School
    Monterey, CA 93943-5100

3. Research Office, Code 09   1
    Naval Postgraduate School
    Monterey, CA 93943-5000

4. Dr. David Hislop   1
    U.S. Army Research Office
    P.O. Box 12211
    Research Triangle Park, NC 27709-2211

5. Dr. Man-Tak Shing, CS/Sh   1
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943

6. Dr. Valdis Berzins, CS/Be   1
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943

7. Dr. Luqi, CS/Lq   7
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943